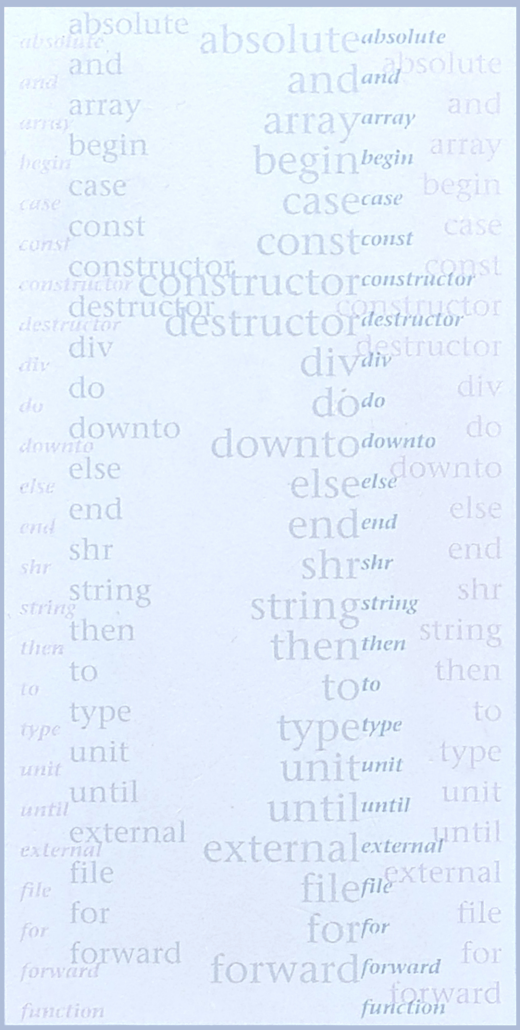
Turbo Pascal 6.0



Objektorientierte Programmierung und Turbo Vision

Das produktive Pascal-Entwicklungssystem für objektorientierte Programmierung. Komplett mit mausgesteuerter Benutzeroberfläche, integriertem Debugger und Assembler.

B O R L A N D

Turbo Pascal 6.0

Turbo Pascal 6.0

Objektorientierte Programmierung und Turbo Vision

B O R L A N D

Copyright © 1990 by Borland International, INC.
Jede Vervielfältigung dieses Handbuches sowie des Turbo Pascal Softwareprogrammes wird strafrechtlich verfolgt. Die Rechte an der Dokumentation und die Rechte an dem Softwareprogramm Turbo Pascal liegen bei Borland International.

Der rechtmäßige Erwerb der Programmdisketten und der Handbücher erlaubt die Nutzung der Programme analog der Benutzung eines Buches. Entsprechend der Unmöglichkeit, daß ein Buch zugleich an verschiedenen Orten von mehreren Personen gelesen wird, darf das Softwareprogramm Turbo Pascal nicht gleichzeitig von verschiedenen Personen an verschiedenen Orten und auf verschiedenen Geräten benutzt werden. Diskettenkopien dürfen lediglich zum Zweck der Datensicherung angefertigt werden.

Sie (der Lizenznehmer von Turbo Pascal) dürfen eigene Programme, die Sie mit dem Turbo-Pascal-Sprachcompiler geschrieben und compiliert haben, ohne Einschränkung oder Verpfichtung zu zusätzlichen Lizenzgebühren verwenden, weitergeben oder verkaufen, solange diese Programme einen Copyright-Vermerk enthalten. Unter Copyright-Vermerk verstehen wir entweder Ihren eigenen Copyright-Vermerk oder gegebenenfalls einen Hinweis wie »Created using © Borland International 1990«. Die Originaldisketten von Turbo Pascal beinhalten einige Dateien mit Hardware- und Zeichensatzinformationen, die von dem Standard-Grafik-Unit (GRAPH.TPU) verwendet werden. Diese Dateien mit den Suffixen .CHR und BGI unterliegen besonderen Nutzungsbeschränkungen. Sie dürfen nicht als Dateien weiterverkauft werden. Sie dürfen diese Dateien aber in Ihren eigenen mit Turbo Pascal geschriebenen und compilierten Programmen verwenden. Sie dürfen diese compilierten Programme auch weiterverkaufen, solange sie einen Copyright-Vermerk beinhalten.

Die Beispielprogramme von Turbo Pascal, die auf den Originaldisketten mitgeliefert werden, demonstrieren die verschiedenen Eigenschaften von Turbo Pascal. Sie dienen ausschließlich zu Lehrzwecken. Sie dürfen diese Beispielprogramme editieren oder für den eigenen Gebrauch modifizieren. Sie dürfen diese Programme jedoch nicht als Ganzes oder in Teilen in compilierter Form oder als Quelltext weitergeben oder verkaufen. Sie dürfen die Routinen aus den Beispielprogrammen in eigenen Anwendungsprogrammen verwenden, solange die resultierenden Programme nicht einem Bespielprogramm oder einem Teil eines Beispielprogrammes in Erscheinung oder Funktion gleichen.

Einschränkung der Gewährleistung
Es wird keine Garantie für die Richtigkeit des Inhaltes dieses Handbuches übernommen. Da sich Fehler, trotz aller Bemühungen, nie vollständig vermeiden lassen, sind wir für Hinweise jederzeit dankbar.

Herausgeber
Borland GmbH
Lindwurmstraße 88
Postfach 15 03 40
8000 München 2
Telefon 0 89 / 72 01 00
Telefax 0 89 / 77 93 39
Telex 5 21 26 37 mcm d

Umschlag und typografische Konzeption: Rudolf Paulus Gorbach

Grafiken: Sid Gastl, Andreas Höhne

Übertragung ins Deutsche: Dieter Strehle, Kapitel 1: Alois Stockklauser

PostScript-Belichtung: CD Computer & Dokumentations GmbH, Aalen

Druck und Bindung: Schoder Druck GmbH, Gersthofen

1. Auflage, Oktober 1990

Inhalt

Einführung 1

Was ist Turbo Vision? 1
Was in den Kapiteln zu finden ist 2
Typographie 3

1. Objektorientiert programmieren 5

Objekte 6
Vererbung 7
Objekte: Rekords, die Eigenschaften vererben 8
Instanzen von Objekttypen 10
Die Felder eines Objekts 11
Guter und schlechter Programmierstil 11
Methoden 12
Code und Daten vereint 13
Die Definition von Methoden 14
Der Gültigkeitsbereich und der Parameter Self 15
Datenfelder eines Objekts und formale Parameter in Methoden 17
Objekte, die von Units exportiert werden 18
Aktiv programmieren 20
Kapselung 21
Methoden: kein zurück 22
Die Erweiterung von Objekten 23
Die Vererbung statischer Methoden 25
Virtuelle Methoden und Polymorphie 27
Statische oder dynamische Bindung 28
Kompatibilität von Objekten 29
Polymorphe Objekte 31
Virtuelle Methoden 33
Ein Beispiel für dynamische Bindung 35
Prozedur oder Methode? 37
Erweiterbarkeit von Objekten 43
Statische oder virtuelle Methoden 46
Dynamische Objekte 47
Speicherplatzreservierung und Initialisierung mit New 47

Entfernen von dynamischen Objekten 48
Destruktoren 49
Ein Beispiel für die dynamische Reservierung von Objekten 51
Wie soll's weitergehen? 57
Zusammenfassung 58

2. Das Rad neu erfinden? 59

Nein – erben! 59
Eine neue Vision der Programmentwicklung 60
Die Elemente eines Turbo-Vision-Programms 61
Views 61
Events 61
Stumme Objekte 62
Ein einfaches Turbo-Vision-Programm 63
Das Menü aktivieren 65
Ein Dialogfenster 66
Schalter 66
Den Dialog abschließen 67
HELLO.PAS von innen 67
Das Objekt TApplication 68
Das Dialogfenster – ein Objekt 69
Programmablauf und Debuggen 70
Das Hauptprogramm von HELLO 70
Die Methode Run 71
Die Methode Done 72

3. Mit Turbo Vision programmieren 73

Ihr erstes Programm 73
Menüleiste, Arbeitsfläche, Statuszeile 75
Die Arbeitsfläche 75
Die Statuszeile 75
Die Menüleiste 77
Wir öffnen ein Fenster 79
Ein Fenster definieren 81
Eigenschaften eines Fensters 82
Innenleben eines Fensters 83
Was zu sehen ist 85
Textdateien betrachten 86
In den Puffer umleiten 87
Das Fenster über den Text bewegen 90
Mehrere Darstellungen in einem Fenster 92
...und wo bringen wir die Produktion unter? 93
Ein Fenster schließen 94
Ein Dialogfenster 94
Einen modalen Dialog definieren 96

Schalter installieren 97
Auswahlfelder 99
Was steht zur Wahl? 101
Das Objekt TInputLine 101
Daten abliefern – Daten zurückgeben 102
Tastenkürzel und ihre Tücken 104
Das Ende eines Dialogs 106
Weitere Dialogelemente 106
Statischer Text 106
Aus Listen wählen 107
Eingaben festhalten 107
Standard-Dialoge 107

4. Die Hierarchie der Objekte 109

Typologie der Objekte 110
Abstrakte Objekte 111
Instanzen definieren oder Objekte ableiten 111
Instanzen definieren 112
Objekte ableiten 112
Die Methoden der Objekte 113
Abstrakte Methoden 113
Pseudo-abstrakte Methoden 113
Virtuelle Methoden 113
Statische Methoden 113
Die Felder der Objekte 114
Noch einmal: Objekte ableiten 115
Konstruktor-Methoden überschreiben. Die Regel... 116
...und die Ausnahme 116
Destruktor-Methoden überschreiben 118
Andere Methoden überschreiben 118
Objekte an der Basis 119
TPoint 119
TRect 120
TObject 120
View-Objekte 120
View-Objekte allgemein 120
Gruppen 121
Terminale View-Objekte 122
Stumme Objekte 125
Stream-Objekte 125
Ressourcen 126
Kollektionen 127
Strings in Listen 128

5. View-Objekte 129

Turbo Vision übernimmt ... 129
Einfache View-Objekte 130
Wo die Darstellung beginnt 130
Wie der Darsteller auftritt 132
Wie der Darsteller agiert 133
Komplexe Darstellungen 134
Gruppen und Subviews 134
Objekte gruppieren 135
Gruppenbild mit Objekten 136
Verwandtschaftsverhältnisse und Dienstverhältnisse 137
Subviews und View-Bäume 139
Selektierte und fokussierte Views 142
Der Selektierungspfad 143
Wer hält den Fokus? 143
Ein Objekt fokussieren 144
Modale View-Objekte 145
Verhaltensänderungen 145
Das Word Options 146
Das Byte GrowMode 148
Das Byte DragMode 149
Der Status eines View-Objekts 150
In welchen Farben ausgeben 151
Farbpaletten 151
Farbpaletten – genauer betrachtet 152
Die Methode GetColor 153
Die Farben ändern 153
Neue Farben hinzufügen 155

6. Ereignisgesteuerte Programme 157

Turbo Vision mit Leben erfüllen 157
Eingaben lesen 157
Das Wesen von Ereignissen 158
Kategorien von Ereignissen 159
Ereignisse und Befehle 161
Ereignisse und ihr Weg 161
...wohin sie gehen 162
Zuständigkeiten 163
Phase: Umleitung von Ereignissen 164
Befehle 166
Befehle definieren 166
Befehle deaktivieren 167
Ereignisse und wie sie behandelt werden 167
Der TEvent-Record 168
Erledigte Ereignisse 170
Liegengelassene Ereignisse 170

Den Event-Mechanismus verändern 170
Zentrale Sammelstelle 171
GetEvent überschreiben 171
Unproduktive Zeiten nutzen 172
Kommunikation zwischen Objekten 172
Zwischenablage 173
Unter uns Objekten gesagt... 173
Wer hört die Botschaft? 174
Hilfe zur Hilfe 175

7. Sichere Programme schreiben 177

Alles oder nichts 177
Die Sicherheitszone 178
Andere Fehler 180
Speicherfresser 181

8. Kollektionen 183

Das Objekt TCollection 183
Dynamische Anpassung der Größe 184
Polymorphe Daten 184
Kollektionen und Typüberprüfung 184
Eine Kollektion erstellen 185
Wiederholmethoden 187
Die Wiederholmethode ForEach 187
Die Wiederholmethoden FirstThat und LastThat 188
Sortierte Kollektionen 189
Strings in einer Kollektion speichern 190
Kollektionen speichern polymorphe Objekte 191
Speicherverwaltung und Kollektionen 194

9. Streams 195

Das Problem: Objekte speichern und laden 195
Die Antwort: Streams 196
Wie Streams zu benutzen sind 197
Den Stream öffnen 197
In den Stream schreiben, aus dem Stream lesen 198
Den Stream schließen 199
Objekte für einen Stream vorbereiten 199
Die Methoden Store und Load 199
Objekte registrieren 200
Registrieren Sie hier 202
Standardobjekte registrieren 202
Wie ein Stream funktioniert 203
Ein Objekt speichern 203

Ein Objekt lesen 203
Objekte, die keine sind 204
Kollektionen und Streams: ein Beispiel 204
Store-Methoden einfügen 205
Die Objekte registrieren 206
In den Stream schreiben 207
Zeiger in einem Stream speichern? 207
Zeiger auf Subviews 208
Auf Kollegen zeigen 209
Die Arbeitsfläche speichern und wieder einlesen 209
Einen Stream kopieren 210
Streams mit wahlfreiemZugriff 210
Daten in einem Stream speichern 211
Ein neues Stream-Objekt definieren 211
Wie Fehler behandelt werden 212

10. Ressourcen 213

Warum Ressourcen verwenden? 213
Was speichert eine Ressourcen-Datei? 214
Eine Ressourcen-Datei anlegen 214
Objekte aus einer Ressourcen-Datei lesen 216
Strings in Listen 217
Eine Liste von Strings erstellen 217

11. Tips und Tricks 219

Die Fehlersuche in Turbo-Vision-Programmen 219
Wo ist es denn abgeblieben? 219
Das haben wir nicht erwartet 220
Das Programm steht 221
Turbo Vision in bestehende Programme einbinden 221
Das Programm durchforsten 222
Ein Programm organisieren 222
Bits manipulieren 223
Flags 224
Bitmasken 224
Bitoperationen 224
Zusammenfassung 225

12. Benutzung der Referenzkapitel 227

Was Sie wo finden 227
Was Sie Namenskonventionen 228

13. Was die Units exportieren 229
Unit Objects 229
Typen 230
Konstanten 230
Variablen 231
Prozeduren und Funktionen 231
Unit Views 232
Typen 232
Konstanten 232
Variablen 236
Routinen 236
Unit Dialogs 236
Typen 236
Konstanten 237
Prozeduren und Funktionen 237
Unit App 237
Typen 238
Variablen 238
Prozedur 238
Unit Menus 238
Typen 238
Prozeduren und Funktionen 239
Unit Drivers 240
Typen 240
Konstanten 240
Variablen 242
Prozeduren und Funktionen 243
Unit TextView 244
Typen 245
Prozedur 245
Unit Memory 245
Variable 245
Prozeduren und Funktionen 245
Unit HistList 246
Variablen 246
Prozeduren und Funktionen 246

14. Verzeichnis der Objekte 247

15. Außerhalb der Objekte 363

Index 429

Einführung

Dieses Handbuch gliedert sich in zwei Teile, von denen der erste Ihnen die objektorientierten Erweiterungen von Turbo Pascal vorstellt und Sie in dieses neue Paradigma des Programmierens einführt. Der zweite Teil behandelt Turbo Vision, einen völlig neuen Zugang zur Entwicklung von Anwendungsprogrammen. Von den insgesamt 15 Kapiteln dieses Bandes sind Turbo Vision 14 Kapitel gewidmet – ein nicht gerade wohlproportioniertes Verhältnis. Aber Turbo Vision ist objektorientiertes Programmieren in der Praxis und stellt daher die passende, wenn auch nicht unkomplizierte Ergänzung zu dem Kapitel dar, das in diese neue Programmiertechnik einführt.

Wir wissen, daß Turbo Vision keine leichte Kost ist, und stellen daher in 10 Kapiteln dar, was Ihnen Turbo Vision bietet und wie Sie von diesen Möglichkeiten Gebrauch machen können. Wir versuchen dabei auch, die Ideen zu erläutern, die hinter der Entwicklung von Turbo Vision standen. Wenn Sie sich die Zeit nehmen, die Prinzipien zu verstehen, die Turbo Vision zugrundeliegen, werden Sie es bald als zeitsparendes und produktives Werkzeug zu schätzen wissen. Die abschließenden Kapitel 12 bis 15 schließlich bieten eine komplette Referenz des von Turbo Vision Gebotenen.

Was ist Turbo Vision?

Turbo Vision stellt ein objektorientiertes Gerüst dar für die Entwicklung von ereignisgesteuerten, fensterorientierten Anwenderprogrammen. Wir haben Turbo Vision geschaffen, um Ihnen die ewige Arbeit zu ersparen, die darin besteht, die immer gleichen Routinen zu entwickeln, auf denen nun einmal jedes Anwendungsprogramm aufbaut. Turbo Vision bietet einiges:

- Einander überlappende, in ihrer Größe und Position änderbare Bildschirmfenster
- Menüverwaltung
- Unterstützung der Maus
- Dialogfenster

- Markierungs- und Schaltfelder, Listen, Eingabefelder...
- ...und vieles mehr... Lassen Sie sich überraschen.

Eigentlich kennen Sie Turbo Vision schon, denn darauf basiert die integrierte Entwicklungsumgebung von Turbo Pascal. Wenn Sie Programme schreiben wollen, die in Erscheinungsweise und Funktionalität dieser Entwicklungsumgebung gleichkommen sollen, dann verwenden Sie Turbo Vision.

Was in den Kapiteln zu finden ist

Kapitel 1 »Objektorientiert programmieren« ist eine Einführung in die wichtigsten Konzepte der objektorientierten Programmierung – die Unterschiede zwischen Objekten und Records, die Vorteile der Kapselung von Daten und Code, Vererbung, Polymorphie, statische Objektinstanzen im Vergleich zu dynamischen. Viele Programmbeispiele veranschaulichen die Prinzipien der objektorientierten Programmierung.

Kapitel 2 »Das Rad neu erfinden?« bietet eine Übersicht darüber, was Turbo Vision zu leisten vermag. Ein Beispielprogramm verschafft Ihnen einen ersten Einblick.

Kapitel 3 »Mit Turbo Vision programmieren« zeigt an Beispielen, wie Sie Turbo Vision in eigenen Programmen einsetzen und die geboteten Möglichkeiten gebrauchen können.

Kapitel 4 »Die Hierarchie der Objekte« beschreibt die Eigenschaften der Objekte und zeigt sie in ihrem Zusammenhang mit den anderen Objekten, die Turbo Vision definiert.

Kapitel 5 »View-Objekte« behandelt das »Äußere« von Turbo-Vision-Programmen: Wie sich Objekte auf dem Bildschirm darstellen und was Sie beachten müssen, wenn Sie ein neues *View*-Objekt ableiten wollen.

Kapitel 6 »Ereignisgesteuerte Programme« behandelt die innere Mechanik eines Turbo-Vision-Programms. Was ist ein Ereignis, welchen Weg nimmt ein Ereignis, wer reagiert wie auf ein Ereignis: all dies ist Gegenstand dieses Kapitels.

Kapitel 7 »Sichere Programme schreiben« führt Sie in die Speicherverwaltung von Turbo Vision ein und hilft Ihnen, Programme zu schreiben, die nicht deshalb abbrechen, weil der Speicher zu klein geworden ist.

Kapitel 8 »Kollektionen« stellt einen neuen, flexiblen Datentyp vor. In einer Kollektion können Sie alles speichern, was in den Speicher paßt.

Kapitel 9 »Stream-Objekte« beschäftigt sich mit der Ein- und Ausgabe von Objekten. Mit einem Stream können Sie Objekte in einer Datei oder im EMS-Speicher ablegen und wieder einlesen.

Kapitel 10 »Ressourcen« entwickelt das Konzept der Kollektion weiter und verbindet eine solche Kollektion mit einem Stream. Ihr Programm kann Objekte aus einer Ressourcen-Datei einlesen, statt sie intern initialisieren zu müssen.

Kapitel 11 »Tips und Tricks« behandelt die spezifischen Probleme, die objektorientierte Programme bieten können, wenn es darum geht, einen Fehler aufzuspüren. In diesem Kapitel finden Sie einige Tips dazu, wie Sie bei der Fehlersuche vorgehen sollten.

Kapitel 12 »Benutzung der Referenzkapitel« zeigt Ihnen, wie die auf dieses Kapitel folgenden Referenzkapitel strukturiert und wie sie zu benutzen sind.

Kapitel 13 »Was die Units exportieren« ist eine nach den Units geordnete Auflistung aller Objekte, Routinen, Datentypen, Variablen und Konstanten, die von den zu Turbo Vision gehörenden Units exportiert werden. Dadurch können Sie ermitteln, welche Unit Sie einbinden müssen, um ein bestimmtes Objekt verwenden zu können.

Kapitel 14 »Verzeichnis der Objekte« führt in alphabetischer Folge alle in Turbo Vision definierten Objekte, einschließlich ihrer Felder und Methoden auf. Jedes Feld und jede Methode ist mit einer ausführlichen Erläuterung versehen.

Kapitel 15 »Außerhalb der Objekte« führt in alphabetischer Folge und mit Erläuterungen versehen alle zu Turbo Vision gehörenden Prozeduren, Funktionen, Datentypen, Variablen und Konstanten auf. Kurz gesagt finden Sie hier alles, was kein Objekt ist.

Typographie

Zur Verdeutlichung von Zusammenhängen verwenden wir in diesem Handbuch mehrere unterschiedliche Schriftarten:

`Schreibmaschine` steht für Syntaxdefinitionen, Eingaben des Benutzers, Ausgaben von Beispielprogrammen und wird für Programmlistings verwendet.

`Schreibmaschine fett` kennzeichnet in Listings reservierte Wörter von Turbo Pascal, zum Beispiel `procedure`, `begin`, `object` usw.

Fettschrift wird im laufenden Text benutzt, um reservierte Wörter (wie **begin**, **end**, **object** usw.) hervorzuheben

Kursivschrift kennzeichnet Namen von Variablen und Bezeichnern im laufenden Text. Außerdem dient diese Schriftart zur Hervorhebung neuer Begriffe.

`[Tasten]` für die Tasten Ihrer Tastatur, zum Beispiel in »Ein Druck auf `[Esc]` entfernt das Menü wieder vom Bildschirm«.

1. Objektorientiert programmieren

Die objektorientierte Programmierung ist eine Programmiermethode, die sich sehr stark daran anlehnt, wie wir alle im täglichen Leben unsere Aufgaben bewältigen. Sie ist in einer natürlichen Evolution aus früheren Innovationen der Entwicklung von Programmiersprachen entstanden: sie ist strukturierter und modularer als frühere Ansätze, die Betonung liegt auf den Konzepten und weniger auf den Details. Drei wesentliche Eigenschaften zeichnen die objektorientierte Programmierung aus:

Kapselung: Durch die Kombination eines Rekords mit Prozeduren und Funktionen, mit deren Hilfe eine solche Struktur bearbeitet wird, ergibt sich ein neuer Datentyp: ein Objekt.

Vererbung: Objekte werden bei der Definition in die Hierarchie eines Objektstammbaums eingegliedert, wobei jedes abstammende Objekt den gesamten Code und alle Daten des Vorfahren übernimmt.

Polymorphie: Eine Funktion erhält eine einzige Bezeichnung, die in der gesamten Objekthierarchie Verwendung findet. Jedes Objekt in der Hierarchie implementiert die Funktion in der für das jeweilige Objekt erforderlichen Art und Weise.

Mit den Spracherweiterungen von Turbo Pascal erhalten Sie die gesamte Leistungsfähigkeit der objektorienterten Programmierung: mehr Struktur und Modularität, mehr Abstraktion und Wiederverwendbarkeit von Modulen. All das ist direkt in der Programmiersprache integriert. Die Kombination dieser Eigenschaften ergibt einen Code, der strukturierter, einfacher erweiterbar und leichter zu pflegen ist.

In mancherlei Hinsicht erfordert die objektorientierte Programmierung von Ihnen, die seit Jahren zum Teil als Standard geltenden Gewohnheiten und Vorstellungen von sauberer Programmierung über Bord zu werfen. Hat man sich aber einmal von den Relikten vergangener Tage befreit, erweist sich die objektorientierte Programmierung als einfach, geradlinig und in vielen Situationen den traditionellen Software-Technologien überlegen.

Eine Anmerkung an diejenigen, die bereits Erfahrungen mit objektorientierter Programmierung in anderen Sprachen gesammelt haben: Legen Sie alle Ihre bisherigen Eindrücke über die objektorientierte Programmierung beiseite und lernen Sie die objektorientierten Besonderheiten von Turbo Pascal ganz unbefangen kennen. Objektorientierte Programmierung ist nicht nur eine einzelne Programmiermethode, vielmehr stellt sie ein ganzes Spektrum neuer Möglichkeiten zur Verfügung. In Bezug auf die in einem Objekt vorliegende

Philosophie ist Turbo Pascal eher mit C++ als mit Smalltalk zu vergleichen. Smalltalk ist ein Interpreter, während Turbo Pascal von Anfang an ein reiner Compiler war. Ein Compiler verarbeitet ein Programm ganz anders (und viel schneller) als ein Interpreter. Turbo Pascal ist als produktives Entwicklungswerkzeug konzipiert und nicht als Hilfsmittel der Forschung.

Eine Anmerkung an diejenigen, die noch nicht wissen, um was es bei der objektorientierten Programmierung geht: Das macht überhaupt nichts. Zuviel Heuchelei, zuviel Verwirrung, und zu viele Leute, die von Dingen reden, von denen sie nichts verstehen, haben in den letzten Jahren viel Staub aufgewirbelt. Versuchen Sie alles zu vergessen, was Ihnen andere über die objektorientierte Programmierung erzählt haben. Die beste Vorgehensweise (im Grunde genommen die *einzige*), um etwas Brauchbares über die objektorientierte Programmierung zu erfahren, besteht darin, das zu tun, was Sie ohnehin gleich machen wollen: Setzen Sie sich hin und probieren Sie es aus.

Objekte

Sehen Sie sich um... Ihr Blick fällt auf den Apfel, den Sie sich zum Mittagessen mitgenommen haben. Machen wir ihn zum Objekt der Betrachtungen darüber, was ein Objekt ist. Angenommen, Sie wollten diesen Apfel formal beschreiben. Das erste, was Sie vielleicht versuchen, ist, ihn auseinanderzunehmen: *S* soll die Oberfläche der Schale beschreiben, *J* das Flüssigkeitsvolumen des enthaltenen Safts, *F* sei das Gewicht der Frucht im Inneren, *D* die Anzahl der Kerne...

Gehen Sie nicht auf diese Art und Weise an die Sache heran. Denken Sie wie ein Maler. Sie sehen einen Apfel und malen diesen Apfel. Das Bild des Apfels ist kein Apfel. Es ist lediglich eine Abbildung auf einer planen Oberfläche. Aber immerhin: das Objekt ist noch als Apfel erkennbar; es sind nicht nur ein paar abstrakte Zahlen. Die Bestandteile stehen in einem Zusammenhang, so daß die wesentlichen Beziehungen zwischen den Einzelteilen erhalten bleiben.

Objekte sind Modelle der Eigenschaften und des Verhaltens von Bestandteilen der uns umgebenden Welt. Sie stellen (bis jetzt) die äußerste Form der Abstraktion dar. Objekte bewahren den Zusammenhang ihrer Eigenschaften und ihres Verhaltens.

Ein Apfel kann zerlegt werden. Ist er aber einmal zerlegt, so ist er kein Apfel mehr. Die Beziehungen der einzelnen Bestandteile zum Ganzen und untereinander sind einfacher zu erfassen, wenn das betrachtete Objekt in seiner Komplexität dargestellt wird. Man spricht dabei von einer *Kapselung*. Dieses Konzept ist ein sehr wichtiger Aspekt der objektorientierten Programmierung. Wir werden bald auf die Kapselung zurückkommen.

Von ebenso großer Bedeutung ist die Tatsache, daß Objekte ihre Eigenschaften und ihr Verhalten von anderen Objekten, ihren *Vorfahren*, *erben* können. Darin besteht der eigentliche qualitative Sprung: Vererbung ist

wohl der entscheidende Unterschied zwischen objektorientiertem Pascal und der bisherigen Programmierung in Turbo Pascal.

Vererbung

Ein Ziel der Wissenschaft ist die Beschreibung der Funktionsweise des Universums. Um sich diesem Ziel zu nähern, investieren die wissenschaftlichen Disziplinen sehr viel Arbeit nur darin, mit den gewonnenen Erkenntnissen einen Stammbaum aufzustellen. Kehren beispielsweise Entomologen aus dem Amazonas mit einem bislang unbekannten Insekt in ihrer Botanisiertrommel zurück, so liegt ein nicht unwesentlicher Aspekt ihrer Arbeit in der Einordnung dieses Insekts in einen Stammbaum, in dem alle bisher bekannten Insektenarten wissenschaftlich klassifiziert und kategorisiert sind. Ähnliche Aufstellungen gibt es für Pflanzen, Fische, Säugetiere, Reptilien, chemische Elemente, Elementarteilchen und Galaxien. An der Spitze der Hierarchie befindet sich eine Hauptkategorie und unter dieser eine wachsende Zahl von Unterkategorien und Unter-Unterkategorien. Diese Abstufung setzt sich solange fort, bis eine weitere Spezifizierung nicht mehr möglich ist.

In der Kategorie *Insekt* gibt es zum Beispiel zwei Unterkategorien: Insekten mit sichtbaren Flügeln und Insekten mit unsichtbaren oder gar keinen Flügeln. Bei den Insekten mit Flügeln gibt es weitere Kategorien: Motten, Schmetterlinge, Fliegen usw. Jede Kategorie hat zahlreiche Unterkategorien, unterhalb deren eine weitere Kategorisierung möglich ist (Abbildung 1.1).

Diesen Klassifizierungsvorgang bezeichnet man als *Taxonomie*. Er veranschaulicht den Mechanismus der Vererbung in der objektorientierten Programmierung.

Die Fragen, die ein Wissenschaftler stellt, wenn er ein neues Tier oder Objekt klassifizieren will, lauten: *In welcher Weise ist es den anderen in einer allgemeinen Klasse ähnlich? Wie unterscheidet es sich?* Jede der verschiedenen Klassen weist eine Reihe von Verhaltensweisen und Eigenschaften auf, durch die seine Klasse festgelegt ist. Ein Wissenschaftler fängt im Stammbaum einer Spezies ganz oben an und steigt dann langsam die einzelnen Verzweigungen herab. Dabei stellt er dann immer wieder seine Fragen. Die höchste Stufe ist am allgemeinsten, dort sind die Fragen am einfachsten: Mit Flügel oder ohne Flügel? In jeder der nachfolgenden Ebenen werden die Fragen detaillierter und damit immer weniger allgemein. Eventuell kommt der Wissenschaftler an einen Punkt, an dem er die Haare des dritten Segments vom Hinterbein des Insekts zählt – hier wird es dann wirklich ganz speziell: vielleicht ein guter Grund, kein Entomologe werden zu wollen.

Der Punkt, auf den es hier ankommt: Ist eine Eigenschaft einmal vorhanden, *verfügen* auch *alle* Kategorien *darunter* über diese Eigenschaft. Sobald Sie also ein Insekt als Mitglied der Ordnung *diptera* (Fliegen) identifizieren, können Sie davon ausgehen, daß dieses spezielle Insekt Flügel besitzt.

Wie Sie bald feststellen werden, ist der Vorgang der objektorientierten Programmierung ähnlich dem Aufbau eines Stammbaums für Datenstruktu-

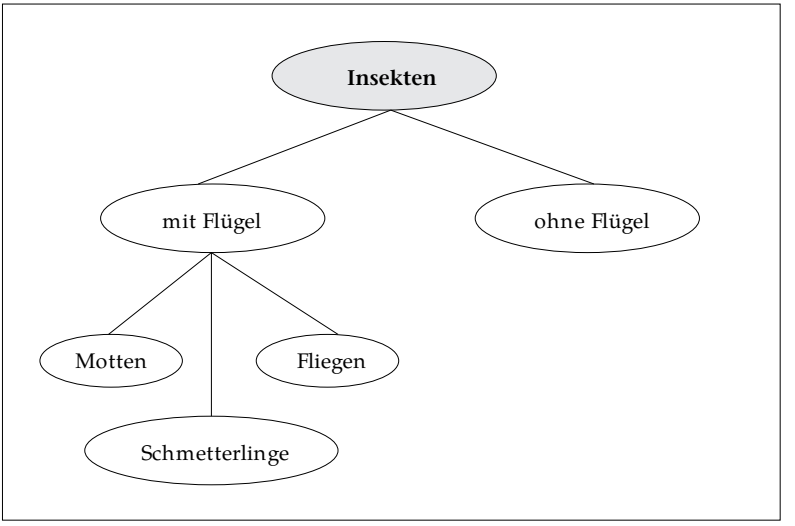


Abbildung 1.1 Ein Ausschnitt aus der Welt der Insekten

ren. Dies ist eines der entscheidenden Konzepte der objektorientierten Programmierung: Die Eigenschaften von einfacheren, allgemeineren Datentypen werden von anderen Datentypen übernommen. Diesen Mechanismus bezeichnen wir als Vererbung.

Objekte: Rekords, die Eigenschaften vererben

In der Begriffswelt von Pascal entspricht ein Objekt weitgehend einem Rekord, der als Verpackung für die Verbindung von mehreren verwandten Datenelementen unter einem Namen dient. Im Zusammenhang mit Grafik können wir die X- und Y-Koordinaten einer Position auf dem Grafikbildschirm zusammenfassen und daraus einen Rekord Location bilden:

```
Location = record
  X, Y : Integer;
end;
```

Der Typ Location ist ein Rekord, d.h. er ist eine Vorlage, die der Compiler dazu verwendet, um Rekord-Variablen zu erzeugen. Eine Variable des Typs Location ist eine Instanz des Typs Location. Der Begriff Instanz taucht hin und wieder in Pascal-Kreisen auf, aber von den Leuten, die sich mit der objektorientierten Programmierung beschäftigen, wird dieser Begriff dauernd verwendet, und auch Sie sollten anfangen, in Begriffen wie Typ und Instanz zu denken.

Mit dem Typ Location haben Sie zwei Interpretationsmöglichkeiten: Wenn

Sie die X- und Y-Koordinaten selbst benötigen, können Sie sich die beiden als Felder X und Y eines Rekords vorstellen. Stellen Sie sich aber die X- und Y-Koordinaten als eine Position auf dem Bildschirm vor, so sollten Sie sich die Koordinaten zusammengefaßt als Location denken.

Nehmen wir einmal an, Sie wollen einen Bildschirmpunkt an einer Stelle, die mit dem Rekord Location beschrieben wird, beleuchten. In Pascal würden Sie ein Boole'sches Feld hinzufügen, das angibt, ob das Pixel leuchtet oder nicht, und daraus einen neuen Rekordtyp bilden:

```
Point = record
  X, Y : Integer;
  Visible : Boolean;
end;7´ 8Abbildung 1.1 Ein Ausschnitt aus der Welt der Insekten
```

Vielleicht gehen Sie noch ein wenig geschickter vor und behalten den Rekordtyp Location bei, indem Sie ein Feld des Typs Location innerhalb des Typs Point aufnehmen:

```
Point = record
  Pos : Location;
  Visible : Boolean;
end;
```

Für Pascal-Programmierer ist das nichts Neues. Eins jedoch bietet diese Vorgehensweise nicht: Sie werden nicht gezwungen, über die Eigenschaften der Objekte nachzudenken, die sie in Ihrem Programm manipulieren. Sie müssen sich Fragen wie die folgende selbst stellen: »Wie unterscheidet sich ein Punkt auf dem Bildschirm von einer Position auf dem Bildschirm?«. Die Antwort lautet: Ein Punkt ist eine Position auf dem Bildschirm, der dargestellt oder nicht dargestellt werden kann. Denken Sie nochmal über den Anfang der Aussage nach: Ein Punkt ist eine Position ... Da haben Sie es.

In dieser Defintion des Punkts ist implizit eine Position für diesen Punkt enthalten. (Ein Pixel ist schließlich nur auf dem Bildschirm vorhanden.) In der objektorientierten Programmierung wird diese spezielle Beziehung besonders berücksichtigt. Da alle Punkte eine Position haben müssen, können wir sagen, daß der Typ Point ein Nachkomme des Typs Location ist. Point erbt alle Eigenschaften, die Location auszeichnen. Gleichzeitig wird in der Definition von Point alles hinzugefügt, was Point an zusätzlichen Eigenschaften haben sollte.

Dieser Vorgang, bei dem ein Typ die Eigenschaften eines anderen Typs erbt, wird als Vererbung bezeichnet. Der Typ, der die Eigenschaften erbt, wird als Nachkommenstyp oder »Nachkomme« bezeichnet. Der Typ, von dem der Nachkomme seine Eigenschaften übernimmt, ist der Vorfahrenstyp oder »Vorfahr«.

Die in Pascal bekannten Rekordtypen können nichts erben. In Turbo Pascal aber ist die Pascal-Sprache so erweitert worden, daß die Vererbung unterstützt wird. Eine dieser Erweiterungen ist eine neue Datenstruktur, die zwar einem Rekord verwandt ist, aber viel mehr Möglichkeiten bietet. Die Datentypen dieser neuen Kategorie werden mit einem neuen reservierten Wort definiert: **object**. Ein Objekttyp kann als abgeschlossener Typ in der Art der Pascal-

Rekords definiert werden, läßt sich aber auch als Nachkomme eines existierenden Objekttyps definieren, indem man den Namen des Vorfahren in Klammern hinter das reservierte Wort **object** schreibt.

Für unser Grafikbeispiel lassen sich die beiden Typen auf folgende Art und Weise definieren:

```
type
  Location = object
    X, Y : Integer;
  end;
  Point = object (Location)
    Visible : Boolean;
  end;
```

Achtung: Die Klammern kennzeichnen die Vererbung der Eigenschaften des Typs Location an den Typ Point.

Location ist in diesem Fall der Vorfahrenstyp und Point der Nachkommenstyp. Wie Sie bald sehen werden, läßt sich dieser Vorgang beliebig oft wiederholen: Ausgehend von Point können Sie weitere Nachkommen definieren und dann wiederum Nachkommen der Nachkommen von Point usw. Ein Großteil des Entwurfs einer objektorientierten Anwendung besteht aus dem Aufbau der Objekthierarchie, in der sich der Stammbaum der Objekte in einer Anwendung ausdrückt.

Alle möglichen Typen, die letztendlich auf das Objekt Location zurückzuführen sind, werden als Nachkommen von Location bezeichnet. Point aber ist einer der direkten Nachkommen. Umgekehrt ist Location der direkte Vorfahr von Point. Ein Objekttyp kann (genauso wie ein Unterverzeichnis in MS-DOS) eine Vielzahl an direkten Nachkommen haben, aber nur einen direkten Vorfahren.

Wie diese Definitionen zeigen, sind Objekte sehr eng mit den Rekords verwandt. Das neue reservierte Wort **object** ist der am meisten ins Auge stechende Unterschied. Wir werden aber noch eine Reihe weiterer Unterschiede, darunter einige sehr interessante, kennenlernen.

Zum Beispiel werden die Felder X und Y von Location im Typ Point nicht explizit angeführt, trotzdem sind diese Felder durch die Vererbung auch in Point vorhanden. Sie können also von einem X-Wert in Point sprechen, so wie Sie von einem X-Wert in Location sprechen können.

Instanzen von Objekttypen

Instanzen von Objekttypen werden genauso deklariert wie alle anderen Variablen in Pascal: entweder als statische Variablen oder als Zeiger auf den Heap:

```
type
  PointPtr = ^Point;
var
```

```
StatPoint : Point;       { sofort einsatzf/hig }
DynaPoint : PointPtr;    { muø zuerst mit New angelegt werden }
```

Die Felder eines Objekts

Auf die Felder eines Objekts können Sie in gleicher Weise zugreifen wie auf die eines normalen Rekords, und zwar entweder über die `with`-Anweisung oder über die Punktmethode. Zum Beispiel:

```
MyPoint.Visible := False;
with MyPoint do
begin
  X := 341;
  Y := 42;
end;
```

Erinnern Sie sich zunächst daran (später wird Ihnen das ganz selbstverständlich vorkommen), daß Sie auf die geerbten Felder genauso zugreifen können wie auf die im gegebenen Objekttyp deklarierten Felder. Auch wenn zum Beispiel X und Y in der Deklaration von Point nicht auftauchen (sie wurden ja vom Typ Location geerbt), können Sie mit ihnen genauso umgehen, als ob sie in Point deklariert worden wären:

```
MyPoint.X := 17;
```

Die ererbten Felder werden genauso behandelt wie die anderen Felder eines Objekts.

Guter und schlechter Programmierstil

Obwohl Sie auf die Felder eines Objekts auch direkt zugreifen können, ist dies nicht die beste Vorgehensweise. Die Prinzipien der objektorientierten Programmierung fordern, die Felder eines Objekts soweit wie möglich in Ruhe zu lassen. Diese Einschränkung mag Ihnen auf den ersten Blick vielleicht als willkürlich und starr erscheinen, sie ist jedoch ein wesentlicher Gesichtspunkt der objektorientierten Programmierung. Mit der Zeit werden Sie den Sinn dieser neuen Definition eines guten Programmierstils erkennen, auch wenn wir vorher, damit alles zusammenpaßt, noch einige Grundlagen besprechen müssen. Im Moment bleibt Ihnen nichts anderes übrig, als uns zu vertrauen: Vermeiden Sie den direkten Zugriff auf die Felder eines Objekts.

Wie aber soll man dann auf die Felder eines Objekts zugreifen? Wie kann man Werte laden und lesen?

Die Antwort lautet: Verwenden Sie die Methoden eines Objekts, um auf die Daten eines Objekts zuzugreifen. Eine Methode ist eine Prozedur oder Funktion, die innerhalb eines Objekts deklariert wird und mit diesem Objekt fest

verbunden ist. Die Datenfelder eines Objekts stehen für die Informationen, die es bereitstellt, seine Methoden beschreiben seine Fähigkeiten.

Methoden

Methoden sind eine der beeindruckendsten Möglichkeiten, die sich durch die objektorientierte Programmierung ergeben. Man braucht jedoch eine Zeitlang, bis man sich daran gewöhnt hat. Am besten lehnen Sie sich zunächst einmal weit zurück und erinnern sich an die Forderung strukturierter Programmierung, daß alle Datenstrukturen initialisiert werden müssen. Betrachten wir dazu die Aufgabe, einen Rekord folgender Definition zu initialisieren:

```
Location = record
  X, Y : Integer;
end;
```

Die meisten Programmierer würden wohl eine **with**-Anweisung verwenden, um den Feldern X und Y einen Wert zuzuweisen:

```
var
  MyLocation : Location;
with MyLocation do
begin
  X := 17;
  Y := 42;
end;
```

Das funktioniert wie erwartet, aber es ist sehr stark an eine bestimmte Instanz des Rekords MyLocation gebunden. Wenn mehr als ein Location-Rekord initialisiert werden muß, benötigen Sie weitere **with**-Anweisungen, um letztlich dasselbe zu machen. Die natürliche Konsequenz ist die Erstellung einer Initialisierungsprozedur, in der die **with**-Anweisung allgemein dazu verwendet werden kann, alle Instanzen des Typs Location über eine Parameterübergabe zu initialisieren:

```
procedure InitLocation(var Target : Location;
                           NewX, NewY : Integer);
begin
  with Target do
  begin
    X := NewX;
    Y := NewY;
  end;
end;
```

Damit wäre die Aufgabe erledigt – wenn Sie aber der Ansicht sind, daß man damit schon ein wenig übertreibt, dann beschleicht Sie wohl dasselbe Gefühl wie die ersten Entwickler in der objektorientierten Programmierung.

Zu diesem Eindruck gehört auch, daß Sie eine Prozedur InitLocation entworfen haben, die nur ganz spezifisch auf den Typ Location angewendet werden kann. Warum muß man eigentlich immer genau angeben, auf welchen Rekordtyp und auf welche Instanz sich InitLocation beziehen soll? Es sollte eine Möglichkeit geben, um den Rekordtyp und den Code für die Initialisierung so zu verschmelzen, daß sich daraus ein nahtloses Ganzes ergibt.

Das läßt sich jetzt ohne weiteres mit Methoden realisieren. Eine Methode ist eine Prozedur oder Funktion, die so eng mit einem bestimmten Typ verschmolzen ist, daß die Methode unausgesprochen in einer **with**-Anweisung steht. In einer Methode kann man direkt auf die Instanzen des entsprechenden Objekts zugreifen. Zur Typdefinition gehört die Kopfzeile der Methode. Die vollständige Definition der Methode enthält zusätzlich den Typbezeichner des Objekts. Der Objekttyp und die Methode sind zwei Gesichter der neuen Datenstruktur, die wir als Objekt bezeichnen:

```
type
  Location = object
    X, Y : Integer;
    procedure Init (NewX, NewY : Integer);
  end;
procedure Location.Init(NewX, NewY : Integer);
begin
  X := NewX; { Das Feld X eines Objekts mit dem Typ Location }
  Y := NewY; { Das Feld Y eines Objekts mit dem Typ Location }
end;
```

Um nun eine Instanz des Typs Location zu initialisieren, rufen Sie einfach dessen Methode auf, so als ob die Methode ein Feld eines Rekords wäre, was sie in gewissem Sinne auch ist:

```
var
  MyLocation : Location;
MyLocation.Init(17, 42); { Ist doch ganz einfach, oder? }
```

Code und Daten vereint

Einer der wichtigsten Lehrsätze der objektorientierten Programmierung ist das Konzept, daß sich der Programmierer Code und Daten während der Programmentwicklung als eine Einheit vorstellen soll. Weder Code noch Daten existieren in einem Vakuum. Daten steuern den Programmablauf und der Code verändert Form und Inhalt von Daten.

Wenn Ihre Daten und Ihr Code getrennte Größen darstellen, dann besteht immer die Gefahr, daß die richtige Prozedur mit den falschen Daten aufgerufen oder die richtigen Daten an die falsche Prozedur übergeben werden. Daß beide übereinstimmen, ist die Aufgabe des Programmierers. Obwohl Pascals

starke Datentypüberprüfung eine gute Hilfe ist, kann damit bestenfalls ausgesagt werden, was absolut nicht zusammenpaßt.

Pascal sagt aber nichts darüber aus, was zusammengehört. Wenn es nicht in einem Kommentar steht, oder wenn Sie es nicht im Kopf haben, dann kann das zu einem Problem werden.

Ein Objekt trägt durch die Zusammenfassung der Deklaration von Code und Daten dazu bei, diese in passender Übereinstimmung zu halten. In einem typischen Fall rufen Sie, um den Wert eines Felds in einem Objekt zu ermitteln, eine Methode auf, die den Wert des gewünschten Felds zurückgibt. Um den Wert eines Felds zu verändern, rufen Sie eine Methode auf, die diesem Feld einen neuen Wert zuweist.

Turbo Pascal zwingt Sie dazu jedoch nicht. Wie die strukturierte Programmierung, so ist auch die objektorientierte Programmierung eine Disziplin, zu der Sie sich selbst zwingen müssen, indem Sie die von der Sprache angebotenen Hilfsmittel verwenden. Turbo Pascal gestattet Ihnen das direkte Lesen und Schreiben von Objektfeldern auch außerhalb einer Methode – aber es unterstützt Sie, guten Programmierstil zu pflegen. So haben Sie die Möglichkeit, über private die auf diese Deklaration folgenden Felder und Methoden zu schützen. Auf solche Felder können dann nur die Methoden des Objekts selbst zugreifen. Private Methoden wiederum können nur von anderen Methoden dieses Objekts aufgerufen werden.

Die Definition von Methoden

Der Vorgang der Definition einer Methode erinnert an die Units von Turbo Pascal. Innerhalb eines Objekts wird eine Methode mit Hilfe des Funktions- oder Prozedurkopfs definiert.

```
type
  Location = object
    X, Y : Integer;
    procedure Init(InitX, InitY : Integer);
    function GetX : Integer;
    function GetY : Integer;
end;
```

Wie Sie sehen, müssen Sie alle Felder vor den Methoden angeben. Diese Regel kennt nur eine Ausnahme. Wenn Sie die private-Deklaration verwenden, teilt sich die Definition Ihres Objekts in zwei parallel strukturierte Blöcke:

```
type
  BeispielObjekt = object
   Feld1 : Word;

    Feld2 : LongInt;
    constructor Init;
    destructor Done;
  private
```

```
    Feld3 : Boolean;
    procedure PrivateMethode;
  end;
```

Die Deklaration private schützt Feld3 vor einem Zugriff von außen: Nur die Methoden von BeispielObjekt können auf dieses Feld zugreifen. Auch die Methode PrivateMethode kann nur von den anderen Methoden, die dieses Objekt besitzt, aufgerufen werden. Beide Blöcke müssen aber die Felder vor den Methoden aufführen.

Wie bei einer Prozedur- oder Funktionsdeklaration im **interface**-Abschnitt einer Unit, sagt die Deklaration einer Methode in einem Objekt aus, was eine Methode macht, aber nicht wie.

Das Wie ist außerhalb der Objektdefinition in einer eigenen Prozedur- oder Funktionsdeklaration definiert. Wenn Methoden außerhalb des Objekts vollständig definiert werden, dann muß vor dem Namen der Methode die Bezeichnung des Objekttyps stehen, zu dem diese Methode gehört. Hinter der Bezeichnung des Objekttyps steht ein Punkt:

```
procedure Location.Init(InitX, InitY : Integer);
begin
  X := InitX;
  Y := InitY ;
end;

function Location.GetX : Integer;
begin
  GetX := X ;
end;

function Location.GetY : Integer;
begin
  GetY := Y;
end;
```

Die Definition von Methoden folgt der gewohnten Punktmethode zur Spezifizierung eines Rekordfelds. Zusätzlich zur Definition von Location.GetX kann man ohne Probleme eine normale Funktion des Namens GetX, dann natürlich ohne die vorangestellte Objektbezeichnung Location, definieren. Die »außerhalb« definierte Funktion GetX hätte natürlich keinerlei Verbindung zum Objekttyp Location.

Der Gültigkeitsbereich und der Parameter Self

Beachten Sie bitte, daß in keiner der bisherigen Methoden eine explizite Konstruktion mit **with** objekt **do** vorgekommen ist. Die Datenfelder eines Objekts sind in den Methoden des Objekts frei verfügbar. Obwohl die Datenfelder und die Methoden eines Objekts getrennt vorliegen, liegen sie im selben Gültigkeitsbereich.

Das ist der Grund, warum eine Methode von Location eine Anweisung wie GetY := Y enthalten kann, ohne daß Y dabei näher spezifiziert werden müßte. Denn Y gehört zu dem Objekt, das von der Methode aufgerufen wurde. Wenn ein Objekt eine Methode aufruft, dann ist unausgesprochen eine Verbindung im Sinne einer Anweisung wie **with** mirselbst **do** Methode zwischen dem Gültigkeitsbereich des Objekts und seinen Methoden hergestellt worden.

Diese implizit vorliegende **with**-Anweisung wird dadurch realisiert, daß ein nicht sichtbarer Parameter bei jedem Aufruf an die Methode übergeben wird. Dieser Parameter hat die Bezeichnung Self und ist letztlich ein kompletter 32-Bit-Zeiger auf die Instanz des Objekts, welche die Methode aufruft. Die Methode GetY, die zu Location gehört, entspricht demnach folgender Funktion:

```
function Location.GetY(var Self: Location) : Integer;
begin
  GetY := Self.Y;
end;
```

Achtung: Dieses Beispiel ist syntaktisch nicht korrekt. Es wurde hier lediglich deshalb angegeben, damit Sie ein besseres Verständnis für die Verbindungen zwischen Objekten und seinen Methoden erhalten.

Ist es für Sie wichtig, daß Sie sich über Self im klaren sind? Normalerweise nicht. Der von Turbo Pascal erzeugte Code behandelt in fast allen Fällen alles automatisch. Es gibt jedoch ein paar Umstände, unter denen Sie in eine Methode tiefer einsteigen und den Parameter Self ausdrücklich einsetzen müssen.

Self ist im Grunde eine automatisch deklarierte Bezeichnung, und wenn Sie in einer Methode Probleme mit Bezeichnern bekommen, dann können Sie diese Schwierigkeiten umgehen, indem Sie den Bezeichner Self bei einem Datenfeld, das zum Objekt der Methode gehört, zusätzlich angeben.

```
type
  MouseStat = record
    Active : Boolean;
    X, Y : Integer;
    LButton, RButton : Boolean;
    Visible : Boolean;
  end;

procedure Location.GoToMouse (MousePos : MouseStat);
begin
  Hide;
  with MousePos do
  begin
    Self.X := X;
    Self.Y := Y;
  end;
  Show;
end;
```

Achtung: Der explizite Einsatz von Self ist durchaus zulässig, aber man sollte die Situationen vermeiden, in denen man den Parameter unbedingt verwenden muß.

Das letzte Beispiel ist notwendigerweise überaus einfach, und die Verwendung von Self hätte man auch einfach durch das Weglassen der **with**-Anweisung in Location.GoToMouse vermeiden können. Sie sind vielleicht bei einer komplizierten Methode einmal in einer Situation, in der Sie mit **with**-Anweisungen die Programmlogik so stark vereinfachen können, daß sich der Einsatz von Self lohnt. Der Self-Parameter ist ein Teil des physikalischen Stack-Bereichs aller Aufrufe von Methoden.

Beachten Sie: Methoden, die in Assembler programmiert werden (ob mit einem externen oder mit dem integrierten Assembler), müssen Self berücksichtigen, wenn Sie auf die Methodenparameter auf dem Stack zugreifen. Weitere Einzelheiten erfahren Sie in den einschlägigen Kapiteln des Bandes Turbo Pascal Programmierhandbuch.

Datenfelder eines Objekts und formale Parameter in Methoden

Aus der Tatsache, daß Methoden und deren Objekte im selben Gültigkeitsbereich liegen, folgt, daß die formalen Parameter einer Methode nicht mit den Datenfeldern des Objekts identisch sein dürfen. Dies ist keine neue Einschränkung, die sich aus der objektorientierten Programmierung ergibt, sondern stimmt mit den bisherigen Regeln für den Gültigkeitsbereich in Pascal überein. Diese Regel besagt dasselbe wie die bekannte Regel, daß formale Parameter einer Prozedur nicht mit einer lokalen Variablen identisch sein dürfen:

```
procedure Komprimiere (Kompresse : MeinDatensatz;
                       Komprimierer, FehlerCode : Integer);
var
  A, B : Char;
  FehlerCode : Integer; { Diese Deklaration f¡hrt zu einem Fehler! }
begin
  ...
```

Die lokalen Variablen und die formalen Parameter einer Prozedur liegen im selben Gültigkeitsbereich und dürfen deshalb nicht identisch sein. Sie erhalten den Fehler 4: *duplicate identifier*, wenn Sie einen Programmcode wie im obigen Beispiel compilieren wollen. Derselbe Fehler tritt auf, wenn Sie einer Methode einen formalen Parameter übergeben wollen, der mit einem Feld im Objekt identisch ist, das die Methode enthält.

Die Umstände sind etwas unterschiedlich, da Prozedurköpfe innerhalb einer Datenstruktur in Turbo Pascal erst mit der Version 5.5 eingeführt wurden, aber die Richtlinien für die Gültigkeitsbereiche in Pascal haben sich nicht geändert.

Objekte, die von Units exportiert werden

Es ist durchaus sinnvoll, Objekte in Units zu definieren. Objekttypen werden im **interface**-Abschnitt der Unit definiert, die Prozedurrümpfe der Prozeduren im **implementation**-Abschnitt. Um Objekte innerhalb einer Unit zu definieren, sind keine speziellen syntaktischen Überlegungen notwendig.

Units können Ihre eigenen privaten Objektdefinitionen innerhalb des **implementation**-Abschnitts enthalten. Solche Typen unterliegen denselben Restriktionen wie alle Datentypen, die im **implementation**-Abschnitt einer Unit definiert sind. Ein Objekttyp, der im **interface**-Abschnitt einer Unit definiert ist, kann Nachkommen im **implementation**-Abschnitt haben. Im Falle, daß Unit B die Unit A verwendet, kann Unit B auch Nachkommen jedes beliebigen Objekttyps definieren, der von Unit A exportiert wird.

Achtung: Exportiert bedeutet, daß ein Programmelement innerhalb des **interface**-Abschnitts einer Unit definiert ist.

Die vorhin beschriebenen Objekttypen und Methoden können in einer Unit folgendermaßen definiert werden:

```
unit Points;
interface
uses Graph;

type
  Location = object
    X, Y : Integer;
    procedure Init(InitX, InitY : Integer);
    function GetX : Integer;
    function GetY : Integer;
  end;

  Point = object (Location)
    Visible : Boolean;
    procedure Init(InitX, InitY : Integer);
    procedure Show;
    procedure Hide;
    function IsVisible : Boolean;
    procedure MoveTo(NewX, NewY : Integer);
  end;

implemenation

{-------------------------------------------------}
{ Implementierung der Methoden von Location       }
{-------------------------------------------------}

procedure Location.Init(InitX, InitY : Integer);
begin
  X := InitX;
```

```
  Y := InitY;
end;

function Location.GetX : Integer;
begin
  GetX := X;
end;

function Location.GetY : Integer;
begin
  GetY := Y;
end;

{--------------------------------------------------}
{ Implementierung der Methoden von Point           }
{--------------------------------------------------}

procedure Point.Init(InitX, InitY : Integer);
begin
  Location.Init(InitX, InitY);
  Visible := False;
end;

procedure Point.Show;
begin
  Visible := True;
  PutPixel(X, Y, GetColor);
end;

procedure Point.Hide;
begin
  Visible := False;
  PutPixel(X, Y, GetBkColor);
end;

procedure Point.IsVisible : Boolean;
begin
  IsVisible := Visible;
end;

procedure Point.MoveTo(NewX, NewY : Integer);
begin
  Hide;
  X := NewX;
  Y := NewY;
  Show;
end;

end.
```

Um die Objekttypen und Methoden anzuwenden, die in der Unit Points definiert sind, brauchen Sie nur die Unit in Ihrem eigenen Programm zu

verwenden und eine Instanz des Typs Point im **var**-Abschnitt Ihres Programms zu deklarieren:

```
program MakePoints;
uses Graph, Points;
var
  APoint : Point;
  ...
```

Zur Erzeugung und Anzeige des Punkts, der durch APoint repräsentiert wird, ruft man lediglich die Methoden von APoint unter Berücksichtigung der Punktsyntax auf:

```
APoint.Init(151, 82);     { Anfangswert f¡r X,Y bei 151, 82. }
APoint.Show;              { APoint schaltet sich ein. }
APoint.MoveTo(163, 101);  { APoint wird nach 163, 101 verschoben. }
APoint.Hide;              { APoint schaltet sich aus. }
```

Da Objekte sehr eng mit Rekords verwandt sind, kann man sie auch innerhalb von **with**-Anweisungen verwenden. In diesem Fall muß man die Bezeichnung des Objekts, in dem die Methode vorliegt, nicht angeben:

```
with APoint do
begin
  Init(151, 82);     { Anfangswert f¡r X,Y bei 151, 82. }
  Show;              { APoint schaltet sich ein. }
  MoveTo(163, 101);  { APoint wird nach 163, 101 verschoben. }
  Hide;              { APoint schaltet sich aus. }
end;
```

Genau wie einen Rekord können Sie auch Objekte als Parameter an Prozeduren übergeben und (wie Sie später sehen werden) auf den Heap legen.

Aktiv programmieren

Das meiste, was bisher über Objekte gesagt wurde, stammt aus der komfortablen Turbo-Pascal-Perspektive, da Sie sehr wahrscheinlich dort Ihre ersten Erfahrungen gesammelt haben. Dies wird sich ändern, wenn wir uns weiter in die Konzepte der objektorientierten Programmierung vorwagen, bei denen die Standard-Pascal-Programmierung immer mehr in den Hintergrund tritt. Hinter der objektorientierten Programmierung steht eine ganz eigene Philosophie, weil sie ihren Ursprung teilweise einer (etwas isolierten) Forschungsgemeinde verdankt, und weil das Konzept so andersartig ist.

Ein oft amüsanter Auswuchs davon ist der seitens der Fanatiker der objektorientierten Programmierung geführte Vergleich von Objekten mit Menschen und menschlichen Eigenschaften. Datenstrukturen sind nicht mehr bloß passive Schubladen, in die Werte hineingesteckt werden. In der neuen Betrachtungsweise der Dinge wird ein Objekt als ein Schauspieler auf einer Bühne betrachtet, der eine Reihe von Versen (Methoden) auswendig weiß.

Wenn Sie ihm (als Regisseur) das Wort geben, rezitiert der Schauspieler Abschnitte aus dem Stück. (Objektorientierte Programmiersprachen wurden aufgrund dieses Vergleichs in der englischen Literatur manchmal auch als »actor languages«, also Schauspielersprachen, bezeichnet.)

Es kann sehr hilfreich sein, sich die Anweisung APoint.MoveTo(242,118) als Auftrag an das Objekt APoint vorzustellen, das »sich selbst zur Position 242,118 begeben soll«. Das Objekt ist das zentrale Konzept. Sowohl die Liste der Methoden als auch die im Objekt enthaltenen Datenfelder sind dem Objekt untergeordnet. Weder der Code noch die Daten haben hier die Oberhand.

Objekte werden nicht deshalb mit Schauspielern auf einer Bühne verglichen, weil sich das recht intelligent anhört. Die Maßgabe der objektorientierten Programmierung versucht sehr stark, die Bestandteile eines Problems als Teil eines Modells umzusetzen und nicht als logische Abstraktionen. Die kleinen Dinge des Alltags, vom Toaster übers Telefon zum Frottierhandtuch, haben alle Eigenschaften (Daten) und Verhaltensweisen (Methoden). Zu den Eigenschaften eines Toasters gehören vielleicht die benötigte Versorgungsspannung, die Anzahl der Scheiben, die auf einmal getoastet werden können, die Einstellung der Toastbräunung, seine Farbe, seine Marke usw. Zu seinen Verhaltensweisen gehören das Aufnehmen der Brotscheiben, das Toasten der Scheiben und das Auswerfen der getoasteten Scheiben.

Wenn wir ein Simulationsprogramm für eine Küche schreiben wollten, was gibt es dann für eine bessere Vorgehensweise, als die verschiedenen Haushaltsgeräte als Objekte umzusetzen, bei denen die Eigenschaften und Verhaltensweisen als Datenfelder und Methoden codiert sind? Das wurde auch schon so gemacht. Die erste objektorientierte Programmiersprache (Simula-67) wurde zur Erstellung solcher Simulationen entwickelt.

Dies ist auch der Grund, warum die objektorientierte Programmierung im konventionellen Sinne so sehr mit grafisch orientierten Benutzeroberflächen in Verbindung gebracht wird. Objekte sollten Simulationen sein, und was gibt es besseres, um ein Objekt zu simulieren, als ein Abbild von ihm zu erstellen? Die Objekte in Turbo Pascal sollen Modellkomponenten des Problems sein, das Sie lösen wollen. Behalten Sie dies im Hinterkopf, wenn Sie sich nun die neuen objektorienterten Erweiterungen von Turbo Pascal anschauen.

Kapselung

Die Verschmelzung von Code und Daten in Objekten wird als Kapselung bezeichnet. Wenn Sie gründlich vorgehen, dann geben Sie genug Methoden an, damit der Anwender niemals mehr direkt auf die Daten des Objekts zugreifen muß. Einige andere objektorientierte Sprachen, wie zum Beispiel Smalltalk, erzwingen die Kapselung, aber in Turbo Pascal haben Sie die Wahl, und für einen guten objektorientierten Programmierstil sind Sie selbst verantwortlich.

Location und Point werden so implementiert, daß es völlig unnötig ist, auf die internen Daten direkt zuzugreifen:

```
type
  Location = object
    X, Y : Integer;
    procedure Init(InitX, InitY : Integer);
    function GetX : Integer;
    function GetY : Integer;
  end;

  Point = object(Location)
    Visible : Boolean;
    procedure Init(InitX, InitY : Integer);
    procedure Show;
    procedure Hide;
    function IsVisible : Boolean;
    procedure MoveTo(NewX, NewY : Integer);
  end;
```

Es gibt hier lediglich drei Datenfelder: X, Y und Visible. Die Methode MoveTo lädt neue Werte in X und Y und die Methoden GetX und GetY geben die Werte von X und Y zurück. Damit besteht kein Grund mehr, direkt auf die Werte von X und Y zuzugreifen. Mit Show und Hide wird der Boolesche Wert Visible zwischen True und False umgeschaltet und die Funktion IsVisible liefert den aktuellen Status von Visible.

Angenommen, es liegt eine Instanz des Typs Point mit der Bezeichnung APoint vor, dann kann man mit den vorhandenen Methoden alle Datenfelder von APoint folgendermaßen indirekt manipulieren:

```
with APoint do
begin
  Init(0, 0);          { Neuen Punkt mit 0,0 initialisieren }
  Show;                { Diesen Punkt anzeigen }
end;
```

Methoden: kein zurück

Wenn man diese Methoden hinzufügt, dann wird der Quellcode von Point ein wenig aufgebläht, aber der intelligente Linker von Turbo Pascal entfernt alle Programmabschnitte der Methoden, die in einem Programm nicht aufgerufen werden. Sie sollten deshalb nicht zögern, eine Methode für einen Objekttyp zu programmieren, die möglicherweise in manchen Programmen gar nicht verwendet wird. Nicht verwendete Methoden beeinflussen die Ausführungsgeschwindigkeit oder den Umfang Ihrer ».EXE«-Datei überhaupt nicht. Wenn die Methoden nicht verwendet werden, dann sind sie gleichsam gar nicht vorhanden.

Hinweis: Es ergeben sich einige enorme Vorteile, wenn man die Möglichkeit hat, Point aus allen globalen Referenzen loszulösen. »Kennt« außerhalb des Objekts niemand die Darstellung der internen Daten, dann kann der Programmierer, der für dieses Objekt verantwortlich ist, die interne Darstellung im Detail so oft ändern wie er will – solange die Deklaration der Methode gleichbleibt.

In manchen Objekten können die Daten als Array dargestellt werden, und später (wenn zum Beispiel der Umfang der Anwendung oder der Daten anwächst) kann man dafür einen binären Baum als effizientere Darstellung in Betracht ziehen. Wenn das Objekt vollständig gekapselt ist, wird durch eine Änderung der Datenstruktur von einem Array in einen binären Baum der Einsatz des Objekts überhaupt nicht berührt. Die Schnittstelle zum Objekt bleibt völlig unverändert, so daß der Programmierer die Implementierung des Objekts verbessern kann, ohne dabei den Code zu ändern, in dem das Objekt verwendet wird.

Die Erweiterung von Objekten

Die meisten Leute, die zum ersten Mal mit Pascal in Kontakt treten, gehen oft von der Flexibilität der Standardprozedur WriteLn aus, bei der man in einer einzigen Prozedur Parameter vieler verschiedener Datentypen verarbeiten kann:

```
WriteLn(CharVar);          { Ausgabe eines Zeichens }
WriteLn(IntegerVar);       { Ausgabe eines Integer-Werts }
WriteLn(RealVar);          { Ausgabe einer Flieøkommazahl }
```

Leider ist es in Standard-Pascal nicht vorgesehen, daß man weitere, ähnlich flexible Prozeduren selbst erstellen kann.

Die objektorientierte Programmierung löst dieses Problem durch Vererbung: Wenn ein Nachkomme definiert ist, dann werden die Methoden des Vorfahren übernommen, aber sie können auch ersetzt werden, falls dies gewünscht wird. Um eine vererbte Methode zu ersetzen, definiert man ganz einfach eine neue Funktion mit demselben Namen der vererbten Funktion, jedoch mit einem anderen Anweisungsteil und (falls dies notwendig ist) mit anderen Parametern.

Ein einfaches Beispiel sollte diesen Vorgang und die damit verbundenen Auswirkungen deutlicher machen. Wir definieren einen Nachkommen von Point, der anstelle eines Punkts einen Kreis auf den Bildschirm zeichnet:

```
type
  Circle = object(Point)
    Radius : Integer;
    procedure Init(InitX, InitY : Integer;
                   InitRadius : Integer);
    procedure Show;
    procedure Hide;
```

```
    procedure Expand(ExpandBy : Integer);
    procedure Contract(ContractBy : Integer);
    procedure MoveTo(NewX, NewY : Integer);
  end;

Procedure Circle.Init(InitX, InitY : Integer;
                      InitRadius : Integer);
begin
  Point.Init(InitX, InitY);
  Radius := InitRadius;
end;

procedure Circle.Show
begin
  Visible := True;
  Graph.Circle(X, Y, Radius);
end;

procedure Circle.Hide
var
  TempColor : Word;
begin
  TempColor := Graph.GetColor;
  Graph.SetColor(GetBkColor);
  Visible := False;
  Graph.Circle(X, Y, Radius);
  Graph.SetColor(TempColor);
end;

procedure Circle.Expand(ExpandBy : Integer);
begin
  Hide;
  Radius := Radius + ExpandBy;
  if Radius < 0 then Radius := 0;
  Show;
end

procedure Circle.Contract(ContractBy : Integer);
begin
  Expand(-ContractBy);
end;

procedure Circle.MoveTo(NewX, NewY : Integer);
begin
  Hide;
  X := NewX;
  Y := NewY;
  Show;
end;
```

Ein Kreis kann in gewissem Sinn auch als ein dicker Punkt betrachtet werden: Er weist alles auf, was auch einen Punkt auszeichnet (eine *X*,*Y*-Position, einen (un)sichtbaren Zustand), er hat aber zusätzlich einen Radius. Der Objekttyp *Circle* scheint nur ein einziges Feld, *Radius*, zu besitzen, aber vergessen Sie nicht, daß *Circle* als Nachkomme alle Felder von *Point* erbt. *Circle* hat also zusätzlich die Felder *X*, *Y* und *Visible*, auch wenn sie nicht in der Definition von *Circle* erscheinen.

Da *Circle* das neue Feld *Radius* definiert, benötigt man zum Initialisieren auch eine neue *Init*-Methode, die sowohl *Radius* als auch die vererbten Felder initialisiert. Anstatt nun direkt eine Zuweisung der Werte an die vererbten Felder, wie *X*, *Y* oder *Visible*, vorzunehmen, können wir auch die Methode von *Point* zur Initialisierung verwenden (wie Sie an der ersten Anweisung von *Circle.Init* erkennen können). Die Syntax für den Aufruf einer vererbten Methode ist *Vorfahr.Methode*, wobei *Vorfahr* den Typbezeichner der vererbten Methode darstellt und *Methode* den Bezeichner der jeweiligen Methode dieses Typs angibt.

Beachten Sie, daß der Aufruf der Methode, die Sie ersetzen wollen, nicht nur ein Beispiel guten Programmierstils ist. Es kann durchaus möglich sein, daß *Point.Init* (oder gar *Location.Init*) irgendeine wichtige, versteckte Initialisierung vornimmt. Indem Sie die Methode, die ersetzt werden soll, aufrufen, stellen Sie sicher, daß der Nachkomme den Funktionsumfang des Vorfahren einschließt. Außerdem werden alle Änderungen, die in den Methoden der Vorfahren vorgenommen werden, automatisch von den Nachkommen übernommen.

Nach dem Aufruf von *Point.Init* kann *Circle.Init* seine eigene Initialisierung durchführen. In diesem Fall ist das nur die Zuweisung des Werts, der im Parameter *InitRadius* übergeben wird, an *Radius*.

Anstatt den Kreis Punkt für Punkt zu zeichnen oder zu löschen, kann man auch einfach die BGI-Prozedur *Circle* verwenden. Wenn Sie jedoch so vorgehen, dann benötigt *Circle* für *Show* und *Hide* auch neue Methoden, die anstelle der Routinen von *Point* verwendet werden. Diese neuen Methoden für *Show* und *Hide* finden Sie im obigen Beispiel im Anschluß an *Circle.Init*.

Mit Hilfe der mit einem Punkt erweiterten Bezeichner kann man Namenskonflikte zwischen BGI-Routinen und Methoden vermeiden. *Graph.Circle* ist eine vollkommen unzweideutige Möglichkeit, um anzugeben, daß Sie die *Circle*-Routine in GRAPH.TPU meinen und nicht das Objekt *Circle*.

Achtung: Während Methoden ersetzt werden können, ist dies bei Datenfeldern nicht der Fall. Sobald ein Datenfeld in einer Objekthierarchie angegeben wurde, darf ein Nachkomme kein Datenfeld gleichen Namens definieren.

Die Vererbung statischer Methoden

In der obigen Definition wird noch eine weitere Methode von *Point* durch eine Methode in *Circle* ersetzt: *MoveTo*. Wenn Sie scharf nachdenken, wundern Sie sich wahrscheinlich, warum *MoveTo* das Feld *Radius* nicht verwendet und warum die Methode keinen Aufruf einer BGI-Routine oder einer anderen

Bibliotheksfunktion zum Zeichnen von Kreisen durchführt. Schließlich wurden die Methoden *GetX* und *GetY* direkt ohne Änderung von *Location* vererbt. Auch *Circle.MoveTo* ist vollkommen identisch mit *Point.MoveTo*. Es wurde nichts anderes verändert, als daß die Bezeichnung *Circle* vor die Bezeichnung *MoveTo* gestellt wurde.

Dieses Beispiel macht ein Problem deutlich, das sich bei Objekten und Methoden stellt, die in dieser Art und Weise definiert sind. Alle Methoden, die wir bisher in Verbindung mit den Objekttypen *Location*, *Point* und *Circle* gesehen haben, waren statische Methoden. (Der Begriff *statisch* bezeichnet diejenigen Methoden, die nicht *virtuell* sind – ein Begriff, den wir in Kürze einführen werden. Virtuelle Methoden sind genau die Lösung dieses Problems, aber um diese Lösung zu verstehen, muß man zuerst die Problematik verstehen.)

Das Problem stellt sich folgendermaßen: Solange keine Kopie von *MoveTo* im Gültigkeitsbereich von *Circle* vorliegt, um die *MoveTo*-Routine von *Point* zu ersetzen, funktioniert diese Routine nicht richtig, wenn sie von einem Objekt des Typs *Circle* aufgerufen wird. Wird die *Point*-Methode *MoveTo* von *Circle* aus aufgerufen, dann wird anstelle des Kreises nur ein Punkt verschoben. Nur wenn das Objekt *Circle* eine Kopie der *MoveTo*-Methode aufruft, die sich in seinem eigenen Gültigkeitsbereich befindet, wird der Kreis mit Hilfe der verschachtelten Aufrufe von *Show* und *Hide* gelöscht und wieder neu gezeichnet.

Warum ist das so? Es hat ein wenig damit zu tun, wie der Compiler den Aufruf einer Methode umsetzt. Wenn der Compiler die Methoden von *Point* compiliert, kommt er zunächst zu *Point.Show* und *Point.Hide* und compiliert die beiden ins Code-Segment. Etwas weiter unten kommt er zur Methode *Point. MoveTo*, die wiederum *Point.Show* und *Point.Hide* aufruft. Wie bei jedem Prozeduraufruf, ersetzt der Compiler die Quellcode-Referenzen auf *Point.Show* und *Point.Hide* mit den Adressen des im Code-Segment erzeugten Codes. Wenn also der Code von *Point.MoveTo* aufgerufen wird, erfolgt dort wiederum ein Aufruf des Codes von *Point.Show* und *Point.Hide*, und alles stimmt überein.

Bis hierher entspricht dies alles der klassischen Vorgehensweise von Turbo Pascal und wäre bereits ab Version 1.0 (außer in Bezug auf die Ausdrucksweise) voll gültig. Die Dinge ändern sich aber, wenn Sie es mit Vererbung zu tun haben. Wenn *Circle* eine Methode von *Point* erbt, dann verwendet *Circle* die Methode genau so, wie sie compiliert wurde.

Schauen Sie sich nochmal an, was *Circle* erbt, wenn *Point.MoveTo* übernommen wird:

```
procedure Point.MoveTo(NewX, NewY : Integer);
begin
  Hide;       { ruft Point.Hide auf }
  X := NewX;
  Y := NewY;
  Show;       { ruft Point.Show auf }
end;
```

Die Kommentare wurden eingefügt, um die Tatsache klarzustellen, daß *Circle* beim Aufruf von *Point.MoveTo* auch *Point.Show* und *Point.Hide* aufruft und nicht *Circle.Show* oder *Circle.Hide*. *Point.Show* zeichnet einen Punkt und keinen Kreis. Solange *Point.MoveTo* die Routinen *Point.Show* und *Point.Hide* aufruft, kann *Point.MoveTo* nicht übernommen werden. Stattdessen muß es durch eine zweite Kopie seiner selbst ersetzt werden, die dann die Kopien von *Show* und *Hide* im richtigen Gültigkeitsbereich aufruft. In diesem Fall sind das *Circle.Show* und *Circle.Hide*.

Die Logik des Compilers arbeitet beim Umsetzen der Aufrufe von Methoden wie folgt: Wenn eine Methode aufgerufen wird, dann sucht der Compiler zuerst nach einer Methode, die im entsprechenden Objekttyp definiert ist. Im Typ *Circle* sind die Methoden mit den Bezeichnungen *Init*, *Show*, *Hide*, *Expand*, *Contract* und *MoveTo* definiert. Wenn eine Methode von *Circle* eine dieser fünf Methoden aufruft, ersetzt der Compiler den Aufruf durch die Adresse einer der *Circle* eigenen Methoden.

Ist keine Methode dieses Namens innerhalb eines Objekttyps definiert, geht der Compiler zum direkten Vorfahren und sucht innerhalb dieses Typs eine Methode mit der entsprechenden Bezeichnung. Findet der Compiler eine Methode dieses Namens, dann tritt die Adresse der Methode des Vorfahren an die Stelle des Bezeichners, an der im Quellcode der Methodenbezeichner des Nachkommens steht. Falls die Suche jedoch erneut erfolglos verläuft, dehnt der Compiler die Suche auf den nächsten Vorfahren aus. Der Compiler setzt die Suche in dieser Art solange fort, bis er beim ersten, in der Objekthierarchie an oberster Stelle stehenden Objekt angelangt ist. Ist die gewünschte Methode auch dort nicht vorhanden, so gibt der Compiler eine Fehlermeldung aus, daß eine derartige Methode nicht definiert ist.

Aber wenn eine vererbte statische Methode gefunden und verwendet wird, dürfen Sie nicht vergessen, daß die aufgerufene Methode genau in der Art und Weise vorliegt, wie sie für den Vorfahren definiert und *compiliert* wurde. Wenn die Methode des Vorfahren andere Methoden aufruft, dann sind damit die Methoden des Vorfahren gemeint, auch wenn der Nachkomme Methoden besitzt, die die Methoden des Vorfahren ersetzen sollen.

Virtuelle Methoden und Polymorphie

Die bisher besprochenen Methoden waren allesamt statische Methoden. Sie sind aus demselben Grund statisch, aus dem auch statische Variablen statisch sind. Der Compiler reserviert den Speicherplatz und löst alle Referenzen mit diesen Variablen *zum Zeitpunkt der Compilierung* auf. Wie Sie gesehen haben, können Objekte und statische Methoden ein wertvolles Hilfsmittel sein, um die Komplexität eines Programms in den Griff zu bekommen.

Manchmal aber ist dies nicht die beste Lösung zur Verarbeitung von Methoden.

Die Probleme, wie sie im vorangegangenen Abschnitt beschrieben wurden, ergeben sich dadurch, daß die Umsetzung der Referenzen auf die Methoden zum Zeitpunkt der Compilierung geschieht. Der Ausweg ist, dynamisch

vorzugehen – und all diese Referenzen zur Laufzeit zu berechnen. Gewisse spezielle Mechanismen müssen vorgesehen sein, damit dies realisiert werden kann. In Turbo Pascal bestehen diese Mechanismen aus der Unterstützung virtueller Methoden.

Achtung: Virtuelle Methoden enthalten ein äußerst mächtiges Hilfsmittel, um eine Verallgemeinerung vorzunehmen, die als Polymorphie bezeichnet wird. *Polymorphie* ist ein griechisches Wort für »viele Ausdrucksformen« und genau das brauchen wir auch: Eine Möglichkeit, um einer Funktion einen Namen zu geben, der in der gesamten Objekthierarchie gleich verwendet werden kann, wobei jedes Objekt in der Hierarchie die Funktion in einer für sie angebrachten Art und Weise implementiert.

Die einfache Hierarchie der Grafikobjekte, die wir bereits beschrieben haben, bietet ein gutes Beispiel für Polymorphie, die mit Hilfe von virtuellen Methoden implementiert werden kann.

Jeder Objekttyp in unserer Hierarchie stellt eine andere Art von Figur auf dem Bildschirm dar: einen Punkt oder einen Kreis. Es ist gewiß sinnvoll zu sagen, daß man einen Punkt oder einen Kreis auf dem Bildschirm darstellen kann. Wenn Sie Objekte definieren, die andere Figuren darstellen sollen, wie zum Beispiel Geraden, Quadrate, Bögen usw., dann können Sie für jede Figur eine Methode schreiben, die das Objekt auf dem Bildschirm anzeigt. Mit der neuen Art des objektorientierten Denkens könnten Sie sagen, daß all diese Grafikfiguren die Fähigkeit besitzen, sich selbst auf dem Bildschirm anzuzeigen. Soviel haben sie alle gemeinsam.

Worin sich die einzelnen Objekttypen unterscheiden, ist die *Art und Weise*, wie sie sich selbst auf dem Bildschirm anzeigen. Ein Punkt wird durch den Aufruf einer Routine zum Zeichnen von Punkten ausgegeben, die nichts weiter als eine *X*,*Y*-Position und vielleicht noch einen Wert für die Farbe braucht. Ein Kreis benötigt eine völlig andere Grafikroutine, um sich selbst darzustellen. Dabei werden nicht nur *X* und *Y* berücksichtigt, sondern auch noch ein *Radius*. Ein Bogen benötigt zudem noch einen Anfangs- und einen Endwinkel und einen viel komplizierteren Zeichenalgorithmus, der diese zusätzlichen Parameter berücksichtigt.

Jedes Grafikobjekt kann gezeichnet werden, aber der Mechanismus, wie es dargestellt wird, ist bei jeder Figur anders. Ein Wort, »Show« wird dazu verwendet, um (buchstäblich) viele Formen wiederzugeben.

Dies ist ein gutes Beispiel für das, was Polymorphie ist und wie virtuelle Methoden in Turbo Pascal realisiert sind.

Statische oder dynamische Bindung

Der Unterschied zwischen dem Aufruf einer statischen und einer dynamischen Methode ist derselbe, der zwischen einer Entscheidung besteht, die man hier und jetzt trifft, und einer, die man auf einen späteren Zeitpunkt verschiebt. Wenn Sie den Aufruf einer statischen Methode codieren, dann sagen Sie dem Compiler im Grunde: »Du weißt, was ich will. Sofort aufrufen«. Beim Aufruf einer virtuellen Methode ist es im Gegensatz dazu so, als ob Sie

sagen würden: »Du weißt doch noch nicht, was ich will. Wenn die Zeit reif ist, schaust du dir die Instanz an«.

Übertragen Sie diesen Vergleich nun auf das Problem mit der Methode *MoveTo*, die wir im letzten Abschnitt erwähnten. Ein Aufruf von *Circle.MoveTo* kann nur an einer Stelle landen: an der am nächsten liegenden Implementierung von *MoveTo* in der Objekthierarchie. In diesem Fall würde *Circle.MoveTo* immer noch die Definition von *MoveTo* in *Point* aufrufen, da *Point* in der Hierarchie als nächstes kommt. Angenommen, kein Nachkomme definiert seine eigene *MoveTo*-Methode, um *MoveTo* von *Point* zu ersetzen, dann hätte jeder Nachkomme dieselbe Implementierung von *MoveTo*. Die Entscheidung kann immer noch zum Zeitpunkt der Compilierung gefällt werden und das ist letztlich alles, was getan werden muß.

Wenn aber *MoveTo* die Methode *Show* aufruft, dann ist das eine ganz andere Geschichte. Jedes Grafikobjekt hat seine eigene Implementierung von *Show*, so daß es lediglich von der Objektinstanz, von der *MoveTo* aufgerufen wurde, abhängig sein sollte, welche Implementierung von *Show* über *MoveTo* aufgerufen wird. Während der Compilierung des Codes von *MoveTo* kann nicht entschieden werden, welche *Show*-Routine aufgerufen werden soll. Die Information darüber ist zum Zeitpunkt der Compilierung nicht verfügbar, so daß die Entscheidung bis zur Laufzeit zurückgestellt werden muß, wenn die Objektinstanz, die *MoveTo* aufruft, abgefragt werden kann.

Den Vorgang, bei dem der Aufruf einer statischen Methode eindeutig mit einer einzigen Methode vom Compiler zum Zeitpunkt der Compilierung aufgelöst werden kann, wird als *statische Bindung* bezeichnet. Einmal hergestellt, ändert sich die Verbindung nicht mehr. Bei der *dynamischen Bindung* können der Aufrufende und der Aufgerufene zum Zeitpunkt der Compilierung nicht miteinander verbunden werden, so daß ein Mechanismus installiert werden muß, der diese Verbindung dann herstellt, wenn der Aufruf dann tatsächlich erfolgt.

Das Wesen dieses Mechanismus ist interessant und sehr ausgeklügelt. In Kürze werden wir sehen, wie er funktioniert.

Kompatibilität von Objekten

Die Vererbung erfordert in Turbo Pascal bei den Regeln zur Kompatibilität der Datentypen einige Änderungen. Zusätzlich zu allen anderen Dingen erbt ein Nachkomme die Kompatibilität zu all seinen Vorfahren. Diese erweiterte Datentypkompatibilität betrifft drei Fälle:

- zwischen Objektinstanzen
- zwischen Zeigern auf Objektinstanzen
- zwischen formalen und aktuellen Parametern

In allen drei Situationen ist es jedoch schwierig, sich zu merken, daß die Typkompatibilität *nur* vom Nachkommen zum Vorfahren geht. Mit anderen Worten: Nachkommen können in beliebiger Art und Weise anstelle der

Vorfahren verwendet werden. Der umgekehrte Weg ist nicht möglich. Betrachten Sie die folgenden Deklarationen:

```
type
  LocationPtr = ^Location;
  PointPtr = ^Point;
  CirclePtr = ^Circle;
var
  ALocation : Location;
  APoint : Point;
  ACircle : Circle;
  PLocation : LocationPtr;
  PPoint : PointPtr;
  PCircle : CirclePtr;
```

Mit diesen Deklarationen sind die folgenden Zuweisungen zulässig:

```
ALocation := Apoint;
APoint := ACircle;
ALocation := ACircle;
```

Einem Vorfahren kann man jeden seiner Nachkommen zuweisen. Die umgekehrten Zuweisungen sind nicht zulässig.

Das ist ein neues Konzept in Pascal und es ist anfangs vielleicht nicht ganz einfach, sich zu merken, in welcher Richtung die Typkompatibilität verläuft. Stellen Sie sich das folgendermaßen vor: *Die Quelle muß immer in der Lage sein, das Ziel vollständig auszufüllen*. Die Nachkommen enthalten aufgrund der Vererbung alles, was zu ihren Vorfahren gehört. Deshalb hat ein Nachkomme entweder genau dieselbe Größe wie sein Vorfahr oder ist (im Normalfall) sogar umfangreicher als seine Vorfahren, niemals aber kleiner. Die Zuweisung eines Vorfahren an einen Nachkommen könnte dazu führen, daß einige Felder des Nachkommens nach der Zuweisung nicht mehr definiert sind. So etwas wäre sehr gefährlich und ist deshalb nicht erlaubt.

In einer Zuweisung dürfen nur die Felder, die in beiden Typen gemeinsam vorhanden sind, von der Quelle zum Ziel kopiert werden. In einer Zuweisung, wie

```
ALocation := ACircle;
```

werden nur die Felder *X* und *Y* von *Acircle* in *ALocation* kopiert, da *X* und *Y* in den Typen *Circle* und *Location* gemeinsam vorkommen.

Die Typkompatibilität ist auch bei Zeigern auf Objekte gegeben, wobei dieselben Regeln wie bei Instanzen von Objekten gelten: Die Zeiger auf Nachkommen können den Zeigern auf Vorfahren zugewiesen werden. Für unsere Beispiele sind folgende Zuweisungen möglich:

```
PPoint := PCircle;
PLocation := PPoint;
PLocation := PCircle;
```

Vergessen Sie nicht, daß die umgekehrten Zuweisungen nicht zulässig sind.

Ein formaler Parameter (entweder ein Werte- oder ein Variablenparameter) eines bestimmten Objekts kann als aktuellen Parameter sich selbst als Objekt oder einen seiner Nachkommen übernehmen. Beim folgenden Prozedurkopf

```
procedure DragIt(Target : Point);
```

sind die Typen *Point* und *Circle*, jedoch nicht der Typ *Location* als aktuelle Parameter zulässig. *Target* hätte auch ein **var**-Parameter sein können.

Achtung: Es gibt jedoch einen gravierenden Unterschied zwischen einem Werte- und einem Variablenparameter: Mit einem Variablenparameter wird ein Zeiger auf das Objekt als Parameter übergeben, während bei einem Werteparameter eine *Kopie* des aktuellen Parameters übergeben wird. Diese Kopie enthält außerdem nur die Felder und Methoden, die im formalen Parametertyp enthalten sind. Das bedeutet, daß der aktuelle Parameter buchstabengetreu in den Typ des formalen Parameters übersetzt wird. Ein Variablenparameter ist eher mit einer Datentypumwandlung vergleichbar, in der die aktuellen Parameter nicht verändert werden.

In ähnlicher Weise kann ein aktueller Parameter ein Zeiger auf dieses Objekt oder ein Zeiger auf alle Nachkommen dieses Objekts sein, wenn als formaler Parameter ein Zeiger auf ein Objekt vorliegt. Mit einem Prozedurkopf wie

```
procedure Figure.Add(NewFigure : PointPtr);
```

können die aktuellen Parameter vom Typ *PointPtr* oder *CirclePtr*, aber nicht vom Typ *LocationPtr* sein.

Polymorphe Objekte

Beim Lesen des vorangegangenen Abschnitts haben Sie sich vielleicht gefragt: Wenn man jeden Nachkommen eines Parametertyps als Parameter übergeben kann, wie kann dann der Anwender des Parameters wissen, welchen Objekttyp er erhält? Der Anwender weiß es tatsächlich nicht, zumindest nicht direkt. Der genaue Typ des aktuellen Parameters ist zum Zeitpunkt der Compilierung unbekannt. Es kann jeder der Nachkommen des **var**-Parametertyps übergeben werden und deshalb wird dieses Objekt auch als *polymorphes Objekt* bezeichnet.

Wozu sollen diese polymorphen Objekte nun gut sein? In erster Linie dazu: *Polymorphe Objekt erlauben die Verarbeitung von Objekten, deren Typ zum Zeitpunkt der Compilierung noch nicht bekannt ist.* Diese Idee ist so neu in Pascal, daß Sie sich darunter wohl nicht sofort ein konkretes Beispiel vorstellen können. (Sie werden jedoch überrascht sein, wie natürlich sich dies ergibt. Das ist dann der Fall, wenn Sie wirklich objektorientiert programmieren.)

Angenommen, Sie sollen eine Grafik-Toolbox schreiben, die eine Reihe von Grafikelementen unterstützen soll: Punkte, Kreise, Quadrate, Rechtecke, Kurven und so weiter. Als Teil dieser Toolbox wollen Sie eine Routine aufnehmen, mit der man ein Grafikelement über den Bildschirm hinweg mit einer Maus verschieben kann.

Beim alten Ansatz hätten wir für jedes Grafikelement, das in der Toolbox unterstützt wird, eine eigene Routine zum Verschieben geschrieben. Sie hätten die Funktionen *DragCircle*, *DragSquare*, *DragRectangle* usw. schreiben müssen. Auch wenn die Typüberprüfung in Pascal es zugelassen hätte (und vergessen Sie nicht, daß es immer eine Möglichkeit gibt, um die Typüberprüfung zu umgehen), dann wären die Unterschiede zwischen den einzelnen Grafikelementen so groß gewesen, daß eine einzige, wirklich allgemeine Routine nicht möglich gewesen wäre.

Schließlich hat ein Kreis einen Radius, aber keine Seiten, ein Quadrat hat nur eine Seitenlänge, ein Rechteck zwei verschiedene Seitenlängen und Kurven, ach...

An diesem Punkt wird nun ein versierter Turbo-Pascal-Spezialist einen Schritt weitergehen und sagen, daß man folgendermaßen vorgehen soll: Übergeben Sie die Grafikelemente in einem Rekord an die Prozedur *DragIt* als Bezugspunkt mit einem allgemeinen Zeiger. Innerhalb von *DragIt* fragen Sie das Auswahlfeld an einem bestimmten Offset im Rekord des Grafikelements ab, um zu bestimmen, welche Art von Grafikelement vorliegt. Danach verzweigen Sie in einer **case**-Anweisung:

```
case FigureIDTag of
  Point     : DragPoint;
  Circle    : DragCircle;
  Square    : DragSquare;
  Rectangle : DragRectangle;
  Curve     : DragCurve;
  ...
```

Wenn man 17 kleine Koffer in einen riesengroßen Koffer packt, dann ist das schon ein kleiner Fortschritt, aber wo liegt das eigentliche Problem, um solche Sachen zu erledigen?

Was passiert, wenn der Anwender der Toolbox einige neue Grafikelemente definieren will? Was ist, wenn der Anwender Ampeln zeichnen will und Achtecke für die Stopschilder braucht? In der Toolbox gibt es keine Achtecke, so daß *DragIt* keine Marke für das Achteck in seiner **case**-Anweisung hätte und sich deshalb weigern würde, das neue Achteck-Element zu verschieben. Wenn man es an *DragIt* übergeben würde, dann würde das Achteck im **else**-Abschnitt der **case**-Anweisung als »unbekanntes Grafikelement« herausfallen.

Die Erstellung von Toolbox-Routinen zum Verkauf ohne Quellcode hat immer mit diesem Problem zu kämpfen: Die Toolbox funktioniert nur mit den Datentypen, die sie »kennt«, die also von den Entwicklern der Toolbox definiert wurden. Der Anwender der Toolbox hat keine Möglichkeit, die Funktionen der Toolbox in einer bestimmten Richtung zu erweitern, ohne daß dies von den Entwicklern vorhergesehen wäre. Was der Anwender gekauft hat, kriegt er auch. Nicht mehr und nicht weniger. Punktum.

Der Ausweg verläuft in die Richtung, daß man die erweiterten Kompatibilitätsregeln in Turbo Pascal anwendet und die eigenen Applikationen unter Zuhilfenahme von polymorphen Objekten und virtuellen Methoden entwickelt. Wenn die Toolbox-Prozedur *DragIt* auf polymorphe Objekte eingerichtet ist, funktioniert sie mit allen Objekten, die in der Toolbox definiert

sind – und allen Nachkommen, die Sie selbst definieren wollen. Falls die Objekte in der Toolbox virtuelle Methoden verwenden, arbeiten die Toolbox-Objekte und -Funktionen mit allen Ihren eigenen speziellen Grafikelementen zusammen. Eine heute von Ihnen definierte virtuelle Methode kann von einer ».TPU«-Datei mit einer Unit der Toolbox aufgerufen werden, die ein Jahr zuvor geschrieben und compiliert wurde. Die objektorientierte Programmierung macht das möglich und die virtuellen Methoden sind der Schlüssel dazu.

Um zu verstehen, wie solche polymorphen Methodenaufrufe durch virtuelle Methoden realisiert werden können, sind einige Hintergrundinformationen zur Deklaration und Anwendung der virtuellen Methoden erforderlich.

Virtuelle Methoden

Eine Methode wird dadurch virtuell, daß man hinter der Deklaration des Objekts das neue reservierte Wort **virtual** angibt. Beachten Sie bitte, daß, wenn Sie eine Methode bei einem Vorfahren mit **virtual** deklarieren, alle Methoden dieses Namens in den Nachkommen auch mit **virtual** deklariert werden müssen, sonst erhalten Sie eine Fehlermeldung des Compilers.

Nachfolgend sehen Sie alle bisher beschriebenen Grafikobjekte in der korrekten virtuellen Deklaration:

```
type
  Location = object
    X, Y : Integer;
    procedure Init(InitX, InitY : Integer);
    function GetX : Integer;
    function GetY : Integer;
  end;

  Point = object(Location)
    Visible : Boolean;
    constructor Init(InitX, InitY : Integer);
    procedure Show; virtual;
    procedure Hide; virtual;
    function IsVisible : Boolean;
    procedure MoveTo(NewX, NewY : Integer);
  end;

  Circle = object(Point)
    Radius : Integer;
    constructor Init(InitX, InitY : Integer;
                     InitRadius : Integer);
    procedure Show; virtual;
    procedure Hide; virtual;
    procedure Expand(ExpandBy : Integer); virtual;
    procedure Contract(ContractBy : Integer); virtual;
  end;
```

Achten Sie zunächst darauf, daß die Methode *MoveTo*, die in der letzten Darstellung des Typs *Circle* stand, nun aus der Typdefinition von *Circle* verschwunden ist. *Circle* benötigt nun keine Ersatzmethode mehr für die *MoveTo*-Routine von *Point*. Stattdessen kann *MoveTo* nun von *Point* vererbt werden, wobei alle verschachtelten Aufrufe von *MoveTo* auf die Methoden von *Circle* zugreifen und nicht auf die Methoden von *Point*, wie dies in einer rein statischen Objekthierarchie der Fall wäre.

Beachten Sie außerdem, daß das neue reservierte Wort **constructor** das reservierte Wort **procedure** in *Point.Init* und *Circle.Init* ersetzt. Ein Konstruktor ist eine spezielle Art von Prozedur, die einige Initialisierungen für den Mechanismus bei virtuellen Methoden erledigt. Des weiteren muß der Konstruktor aufgerufen werden, bevor eine virtuelle Methode aufgerufen wird. Ohne vorangegangenen Aufruf eines Konstruktors kann der Aufruf einer virtuellen Methode zum Absturz des Systems führen. Der Compiler hat jedoch keine Möglichkeit, die Reihenfolge zu überprüfen, in der die Methoden aufgerufen werden.

Jede einzelne Instanz eines Objekts muß mit einem eigenen Konstruktoraufruf initialisiert werden. Es reicht nicht, eine Instanz eines Objekts zu initialisieren und diese Instanz dann den anderen Instanzen zuzuweisen. Die zusätzlichen Instanzen werden durch die Zuweisungen nicht initialisiert, obwohl sie vielleicht sogar die richtigen Daten enthalten. Beim Aufruf der virtuellen Methoden stürzt das System ab.

Was wird in den Konstruktoren gemacht? Jedes Objekt hat eine *virtuelle Methodentabelle* (VMT) im Datensegment. In der VMT ist die Größe des Objekts gespeichert, und für jede virtuelle Methode ist ein Zeiger auf den Code enthalten, mit dem die Methode implementiert wurde. Was nun der Konstruktor macht, ist folgendes: Er stellt eine Verbindung zwischen der Instanz, die den Konstruktor aufruft, und der VMT des Objekts her.

Folgendes sollten Sie sich unbedingt merken: Jedes Objekt hat nur eine virtuelle Methodentabelle. Einzelne Instanzen eines Objekts (also Variablen dieses Objekts) enthalten eine Verbindung zur VMT – sie enthalten nicht die VMT selbst. Der Konstruktor bestimmt den Wert dieser Verbindung zur VMT – was auch der Grund dafür ist, daß sich Ihr Rechner ins Niemandsland verabschiedet, wenn Sie eine virtuelle Methode vor dem jeweiligen Konstruktor aufrufen.

Bereichsüberprüfung beim Aufruf virtueller Funktionen

Während der Programmentwicklung werden Sie wahrscheinlich die Vorteile der Sicherheitsvorkehrungen, die Turbo Pascal bei Aufrufen virtueller Methoden anbietet, ausnützen wollen. Wenn der Schalter *$R* eingeschaltet ist, {*$R+*}, dann wird bei allen Aufrufen von virtuellen Methoden der Initialisierungsstatus der Instanz überprüft, die einen Aufruf ausführt. Wurde die Instanz, die den Aufruf durchführt, noch nicht initialisiert, kommt es zur Laufzeit zu einem Bereichsfehler.

Wenn Sie ein Programm ausgetestet haben und sicher sind, daß keine Methodenaufrufe von nicht initialisierten Instanzen mehr vorhanden sind, können Sie Ihr Programm etwas schneller machen, indem Sie die Option *$R*

ausschalten, {*$R-*}. Methodenaufrufe von nicht initialisierten Instanzen werden dann nicht mehr überprüft und führen möglicherweise zu einem Systemabsturz.

Einmal virtuell – immer virtuell

Sie werden bemerken, daß sowohl *Point* als auch *Circle* Methoden des Namens *Show* und *Hide* besitzen. Alle Methodendeklarationen für *Show* und *Hide* sind mit Hilfe des reservierten Worts **virtual** als virtuelle Methoden gekennzeichnet worden. Sobald ein Vorfahr eine Methode mit **virtual** versieht, müssen alle Nachkommen, die eine Methode dieses Namens implementieren, diese Methode ebenfalls mit **virtual** angeben. Mit anderen Worten, eine statische Methode kann niemals eine virtuelle Methode ersetzen. Wenn Sie es trotzdem probieren, ergibt dies eine Fehlermeldung des Compilers.

Sie sollten sich also merken, daß die Methodendeklaration beim Abstieg in der Objekthierarchie in *keinerlei* Weise verändert werden darf, sobald eine Methode einmal virtuell vorliegt. Sie können sich jede einzelne Definition einer virtuellen Methode als eine Art Tor zu *allen* anderen vorstellen. Aus diesem Grund müssen die Kopfzeilen aller Implementierungen derselben virtuellen Methode identisch sein, bis hin zur Anzahl und dem Typ der Parameter. Bei statischen Methoden ist das nicht der Fall. Eine statische Methode, die eine andere statische Methode ersetzt, darf eine unterschiedliche Anzahl und andere Typen von Parametern verwenden, falls dies angebracht erscheint.

Ein Beispiel für dynamische Bindung

Um zu zeigen, wie man polymorphe Objekte mit dynamischer Bindung in einem Turbo-Pascal-Programm verwendet, kehren wir wieder zur Unit mit den Grafikelementen zurück, die wir bereits früher beschrieben haben. Das Ziel besteht darin, eine Unit zu erzeugen, die mehrere Grafikobjekte (wie *Point* und *Circle*) exportiert und eine allgemeine Routine zum Verschieben dieser Elemente auf dem Bildschirm bietet. Diese Unit, *Figures* genannt, wird eine einfache Implementierung der vorhin besprochenen Grafik-Toolbox sein. Um zu zeigen, wie *Figures* verwendet werden kann, erstellen wir ein einfaches Programm, welches ein neues Grafikobjekt definiert, das in *Figures* nicht vorhanden ist. Dann verwenden wir virtuelle Methoden, um dieses neue Element auf dem Bildschirm zu bewegen.

Überlegen Sie einmal, worin sich die Grafikelemente gleichen und wodurch sie sich unterscheiden. Die Unterschiede sind offensichtlich und alle beziehen sich auf Formen, Winkel und Kurven, die auf den Bildschirm gezeichnet werden. In dem einfachen Grafikprogramm, das wir hier skizzieren, werden sich die auf dem Bildschirm darstellbaren Elemente durch folgende Eigenschaften auszeichnen:

- Sie haben eine Position, die in *X*, *Y* angegeben wird. Der Punkt innerhalb

der Figur, mit der die *X*,*Y*-Position angegeben wird, wird als Ursprung des Elements bezeichnet.

- Alle Elemente können entweder sichtbar oder unsichtbar vorliegen. Dies wird durch eine Boolesche Variable mit den Werten *True* (sichtbar) oder *False* (unsichtbar) angegeben.

Wenn Sie sich die früheren Beispiele ins Gedächtnis rufen, werden Sie feststellen, daß dies genau die Eigenschaften der Objekte *Location* und *Point* sind. *Point* ist im Grunde genommen der Urvater aller Grafikobjekte, die davon abgeleitet werden. Dieser Gedanke zeigt ein wichtiges Prinzip der objektorientierten Programmierung: Durch die Definition einer Objekthierarchie werden alle allgemeinen Attribute in einem einzigen Typ vereint und über diese Hierarchie können die allgemeinen Eigenschaften dieses Typs an alle anderen Typen vererbt werden.

Der Typ *Point* dient als eine Vorlage, aus der alle seine Nachkommen die gemeinsamen Elemente übernehmen können. In diesem Beispiel wird überhaupt kein Objekt des Typs *Point* auf dem Bildschirm ausgegeben, obwohl dies ohne weiteres möglich wäre. (Ruft man *Point.Show* auf, wird natürlich ein Punkt auf den Bildschirm gezeichnet.) Einen Objekttyp, der speziell dafür entwickelt wurde, um vererbbare Eigenschaften für seine Nachfolger bereitzustellen, bezeichnen wir als einen *abstrakten* Objekttyp. Bei einem abstrakten Typ kommt es auf die Nachkommen an und nicht auf die Instanzen.

Gehen Sie nochmal ein paar Seiten zurück und schauen Sie sich *Point* erneut an – dieses Mal aber als eine Zusammenstellung aller Eigenschaften, die alle Grafikobjekte gemeinsam haben. *Point* erbt *X* und *Y* vom vorher definierten Typ *Location*, aber *Point* enthält auch *X* und *Y* und kann sie daher an seine Nachkommen weitergeben. Beachten Sie, daß in keiner der Methoden von *Point* auf die Form eines Grafikelements Bezug genommen wird, daß aber alle Grafikelemente sichtbar oder unsichtbar sein und auf dem Bildschirm hin und her bewegt werden können.

Point hat außerdem noch eine wichtige Funktion als »Verteiler« von Änderungen in der Objekthierarchie als *Ganzes*. Wenn irgendeine neue Eigenschaft ins Spiel kommt, die in allen Grafikelementen berücksichtigt werden soll (Farbe zum Beispiel), kann sie an alle anderen Objekttypen weitergegeben werden, die Nachkommen von *Point* sind, indem man die neuen Eigenschaften in *Point* hinzufügt. Die neuen Eigenschaften sind sofort in allen Nachkommen von *Point* verfügbar. Eine Methode, die ein Element zur aktuellen Position des Mauszeigers verschiebt, kann man zum Beispiel ohne Änderung der elementspezifischen Methoden in *Point* deswegen einfügen, da sich solch eine Methode nur auf die beiden Felder *X* und *Y* auswirkt.

Wenn diese neue Eigenschaft in den einzelnen Grafikelementen verschieden implementiert wird, muß eine ganze Reihe elementspezifischer virtueller Methoden in der Hierarchie eingefügt werden, wobei jede Methode die des direkten Vorfahren ersetzt. Farbe erfordert zum Beispiel nur geringfügige Änderungen in den Methoden *Show* und *Hide*, da die Syntax der Zeichenroutinen von GRAPH.TPU nur davon abhängt, wie die Zeichenfarbe angegeben wird.

Prozedur oder Methode?

Ein Hauptziel der Entwicklung der Unit FIGURES.PAS ist, daß die Anwender der Unit die Objekte, die in dieser Unit definiert sind, erweitern und trotzdem alle Möglichkeiten der Unit ausschöpfen können. Es ist eine interessante Herausforderung für jemanden, der ein Hilfsmittel zum benutzergesteuerten Verschieben eines Grafikelements auf dem Bildschirm entwickeln will.

Es gibt zwei Möglichkeiten, dies zu erreichen. Die Möglichkeit, die einem traditionellen Pascal-Programmierer zuerst einfällt, wäre, daß FIGURES.PAS eine Prozedur exportiert, die ein polymorphes Objekt als **var**-Parameter übernimmt und dieses Objekt dann auf dem Bildschirm umherschiebt. So eine Prozedur finden Sie nachfolgend:

```
procedure DragIt(var AnyFigure : Point; DragBy : Integer);
var
  DeltaX, DeltaY : Integer;
  FigureX, FigureY : Integer;
begin
  AnyFigure.Show; { Element darstellen, das verschoben werden soll }
  FigureX := AnyFigure.GetX;           { Anfangsposition ermitteln }
  FigureY := AnyFigure.GetY;
  { Dies ist nun die Schleife zum Verschieben des Grafikelements. }
  while GetDelta(DeltaX, DeltaY) do
  begin                { Verschiebung auf Element in X,Y anwenden }
    FigureX := FigureX + (DeltaX * DragBy);
    FigureY := FigureY + (DeltaY * DragBy);
    AnyFigure.MoveTo(FigureX, FigureY);      { Element verschieben }
  end;
end;
```

DragIt ruft eine zusätzliche Routine, *GetDelta*, auf, die irgendeine Änderung der Werte von *X* und *Y* vom Anwender übernimmt. Dies kann über die Tastatur, eine Maus oder einen Joystick geschehen. (Der Einfachkeit halber wird die Eingabe in unserem Beispiel über die Pfeiltasten entgegengenommen.)

Was man sich bei *DragIt* unbedingt merken sollte, ist die Tatsache, daß jedes Objekt des Typs *Point* und alle von *Point* abstammenden Typen im **var**-Parameter *AnyFigure* übergeben werden können. Die Instanzen von *Point* oder *Circle* sowie alle zukünftig definierten Typen, die von *Point* oder *Circle* abstammen, können ohne Schwierigkeiten in *AnyFigure* übergeben werden.

Wie aber weiß der Code in *DragIt*, welcher Objekttyp nun tatsächlich übergeben wurde? Er hat überhaupt keine Ahnung – und das ist gut so. *DragIt* bezieht sich nur auf die Bezeichner, die im Typ *DragIt* definiert sind. Durch die Vererbung sind diese Bezeichner auch in allen Nachkommen des Typs *Point* definiert. Die Methoden *GetX*, *GetY*, *Show* und *MoveTo* sind genauso im Typ *Circle* vorhanden, wie sie im Typ *Point* definiert sind, und wären auch in allen zukünftig als Nachkommen definierten Objekten vorhanden.

GetX, *GetY* und *MoveTo* sind statische Methoden. Das bedeutet, daß *DragIt* die Adresse der einzelnen Prozeduren zum Zeitpunkt der Compilierung kennt. Im Gegensatz dazu ist *Show* eine virtuelle Methode. Es gibt jeweils eine andere Implementierung von *Show* in *Point* und *Circle*. Zum Zeitpunkt der Compilierung weiß *DragIt* nicht, welche Implementierung aufgerufen werden soll. Kurzgesagt, wenn *DragIt* aufgerufen wird, dann sucht *DragIt* die Adresse der korrekten Implementierung von *Show* in der VMT der Instanz, die in *AnyFigure* übergeben wurde. Wenn als Instanz *Circle* vorliegt, ruft *DragIt* die Methode *Circle.Show* auf. Wenn als Instanz *Point* vorliegt, ruft *DragIt* die Methode *Point.Show* auf. Die Entscheidung, welche Implementierung von *Show* aufgerufen werden soll, wird bis zur Laufzeit verschoben und solange nicht getroffen, bis das Programm in *DragIt* die virtuelle Methode *Show* aufruft.

DragIt funktioniert also ganz ausgezeichnet. Wenn es von der Toolbox-Unit exportiert wird, kann man damit jeden beliebigen Nachkommen von *Point* auf dem Bildschirm hin und her bewegen, wobei es keine Rolle spielt, ob der Typ bereits während der Compilierung der Toolbox vorhanden war oder nicht. Aber Sie sollten noch einen Schritt weiter denken: Wenn jedes Objekt auf dem Bildschirm hin und her bewegt werden kann, warum sollte man diese Verschiebung nicht zu einer Eigenschaft des Grafikobjekts selbst machen? Mit anderen Worten: Warum soll man aus *DragIt* keine Methode machen?

Warum eigentlich nicht? Wieso sollte man ein Objekt an eine Prozedur übergeben, um dieses Objekt dann auf dem Bildschirm hin und her zu bewegen? Das ist ein alter Hut. Wenn man eine Prozedur schreiben kann, die alle Grafikobjekte auf dem Bidschirm hin- und herbewegt, dann sollten die Grafikobjekte auch selbst in der Lage sein, sich auf dem Bildschirm hin und her zu bewegen. Mit anderen Worten: Anstelle einer Prozedur *DragIt* sollte man eigentlich die Methode *Drag* verwenden.

Eine neue Methode in eine bestehende Objekthierarchie einzuführen, erfordert ein wenig Überlegung. Wie weit oben sollte die Methode in der Hierarchie eingefügt werden? Denken Sie über die Nützlichkeit der Methode nach und entscheiden Sie dann, inwieweit diese Hilfsfunktion anwendbar ist. Zum Verschieben einer Figur gehört die Änderung der Position dieser Figur als Antwort auf die Eingabe des Anwenders. Metaphorisch ausgedrückt, können Sie sich die Methode *Drag* als ein *MoveTo* mit einer eingebauten Energiequelle vorstellen. Im Sinne der Vererbung ist die Funktion direkt neben *MoveTo* einzuordnen. Jedes Objekt, auf das man *MoveTo* anwenden kann, sollte auch *Drag* erben. *Drag* sollte daher in unserem abstrakten Objekttyp *Point* aufgenommen werden, damit dieses Objekt in allen Nachkommen von *Point* vorhanden ist.

Muß *Drag* **virtual** deklariert werden? Der Lackmus-Test, um festzustellen, ob man eine Methode virtuell anlegen sollte, besteht aus der Frage, ob sich der Funktionsumfang dieser Methode irgendwann im Verlauf des Hierarchiebaums ändern wird oder nicht. *Drag* hat eine in sich abgeschlossene Funktion: Die Methode verändert lediglich die *X,Y*-Position einer Figur und es ist kaum denkbar, daß in dieser Funktion mehr als dies behandelt werden soll. Deshalb braucht man sie wahrscheinlich nicht virtuell anzulegen.

Seien Sie bei jeder dieser Entscheidungen sehr vorsichtig: Wenn Sie *Drag* nicht virtuell anlegen, dann nehmen Sie dem Anwender von FIGURES.PAS

bei dieser Prozedur jede Möglichkeit, FIGURES.PAS zu erweitern. Sie können sich die Umstände eventuell gar nicht vorstellen, unter denen ein Anwender *Drag* neu schreiben will. Das bedeutet keineswegs, daß sich solche Umstände nicht ergeben könnten.

Drag hat zum Beispiel eine Eigenschaft, die daraufhin hinweist, daß es vielleicht doch besser ist, *Drag* virtuell zu deklarieren: Die Methode hat etwas mit der *Behandlung von Ereignissen* zu tun, also der Unterbrechung der Eingabe durch Geräte wie der Tastatur und der Maus, deren Zeitpunkt nicht vorhersehbar ist, die aber trotzdem sofort nach dem Auftreten behandelt werden müssen. Eine Ereignisbehandlung ist eine sehr schwierige Sache, die zumeist auch stark von der jeweiligen Hardware abhängig ist. Wenn Ihr Anwender ein Eingabegerät verwendet, das in *Drag* nicht richtig berücksichtigt wurde, hat der Anwender keine Möglichkeit mehr, *Drag* neu zu schreiben. Brechen Sie nicht alle Brücken hinter sich ab. Machen Sie *Drag* virtuell.

Die Schritte, die notwendig sind, um *DragIt* in eine Methode umzuwandeln und in *Point* einzufügen, sind fast trivial. Innerhalb des Objekts *Point* wird *Drag* einfach mit einem anderen Methodenkopf deklariert:

```
Point = object(Location)
  Visible : Boolean;
  constructor Init(InitX, InitY : Integer);
  destructor Done; virtual;
  procedure Show; virtual;
  procedure Hide; virtual;
  function IsVisible : Boolean;
  procedure MoveTo(NewX, NewY : Integer);
  procedure Drag(DragBy : Integer); virtual;
end;
```

Die Position der Methodendeklaration in der Objektdefinition von *Point* spielt keine Rolle. Wie Sie wissen, können die Methoden in jeder beliebigen Reihenfolge deklariert werden, aber die Datenfelder *müssen* vor der ersten Methodendeklaration stehen.

Die Änderung der Prozedur *DragIt* in die Methode *Drag* ist fast genau dasselbe, als würde man den Gültigkeitsbereich von *Point* auf *DragIt* ausdehnen. In der Prozedur *DragIt* mußte man *AnyFigure.Show*, *AnyFigure.GetX* usw. angeben. Da *Drag* nun zu *Point* gehört, braucht man die vollständigen Methodenbezeichner nicht mehr anzugeben. *AnyFigure.GetX* ist nun einfach *GetX* usw. Und natürlich verschwindet auch der **var**-Parameter *AnyFigure* aus der Parameterzeile. Über den implizit übergebenen Parameter *Self* weiß man nun, welche Instanz *Drag* aufgerufen hat.

Nachfolgend sehen Sie den vollständigen Quellcode von FIGURES.PAS zusammen mit *Drag* als virtueller Methode:

```
unit Figures;   { virtuelle Methoden und polymorphe Objekte }

interface
uses Graph, Crt;
```

```
type
  Location = object
    X, Y : Integer;
    procedure Init(InitX, InitY : Integer);
    function GetX : Integer;
    function GetY : Integer;
  end;

  PointPtr = ^Point;
  Point = object(Location)
    Visible : Boolean;
    constructor Init(InitX, InitY : Integer);
    destructor Done; virtual;
    procedure Show; virtual;
    procedure Hide; virtual;
    function IsVisible : Boolean;
    procedure MoveTo(NewX, NewY : Integer);
    procedure Drag(DragBy : Integer); virtual;
  end;

  CirclePtr = ^Circle;
  Circle = object(Point)
    Radius : Integer;
    constructor Init(InitX, InitY : Integer;
                     InitRadius : Integer);
    procedure Show; virtual;
    procedure Hide; virtual;
    procedure Expand(ExpandBy : Integer); virtual;
    procedure Contract(ContractBy : Integer); virtual;
  end;

implementation

{-----------------------------------------------------------------------}
{ Implementierung der Methoden von Location                             }
{-----------------------------------------------------------------------}

procedure Location.Init(InitX, InitY : Integer);
begin
  X := InitX;
  Y := InitY;
end;

function Location.GetX : Integer;
begin
  GetX := X;
end;

function Location.GetY : Integer;
begin
```

```
  GetY := Y;
end;

{-------------------------------------------------------------------}
{ Implementierung der Methoden von Point                            }
{-------------------------------------------------------------------}

constructor Point.Init(InitX, InitY : Integer);
begin
  Location.Init(InitX, InitY);
  Visible := False;
end;

destructor Point.Done;
begin
  Hide;
end;

procedure Point.Show;
begin
  Visible := True;
  PutPixel(X, Y, GetColor);
end;

procedure Point.Hide;
begin
  Visible := True;
  PutPixel(X, Y, GetBkColor);
end;

function Point.IsVisible : Boolean;
begin
  IsVisible := Visible;
end;

procedure Point.MoveTo(NewX, NewY : Integer);
begin
  Hide;
  X := NewX;
  Y := NewY;
  Show;
end;

function GetDelta(var DeltaX : Integer;
                  var DeltaY : Integer) : Boolean;
var
  KeyChar : Char;
  Quit : Boolean;
begin
  DeltaX := 0; DeltaY := 0; { 0 bedeutet keine Positions/nderung. }
  GetDelta := True ;        { True bedeutet, daø eine Verschiebung }
                            { stattgefunden hat. }
```

```
  repeat
    KeyChar := ReadKey;     { zun/chst einen Tastendruck lesen }
    Quit := True;           { von einer zul/ssigen Taste ausgehen }
    case Ord(KeyChar) of
      0: begin          { 0 zeigt einen erweiterten 2-Byte-Code an }
           KeyChar := ReadKey; { Zweites Byte des Codes lesen }
           case Ord(KeyChar) of
            72: DeltaY := -1; { Pfeil nach oben; Y dekrementieren }
            80: DeltaY :=  1; { Pfeil nach unten; Y inkrementieren }
            75: DeltaX := -1; { Pfeil nach links; X dekrementieren }
            77: DeltaX :=  1; { Pfeil nach rechts; X inkrementieren}
             else Quit := False; { alle anderen Tasten ¡bergehen }
           end; { case }
         end;
      13: GetDelta := False; { CR bedeutet, daø keine }
                             { Verschiebung durchgef¡hrt wird }
      else Quit := False;    { alle anderen Tasten ¡bergehen }
    end; { case }
  until Quit;
end;

procedure Point.Drag(DragBy : Integer);
var
  DeltaX, DeltaY : Integer;
  FigureX, FigureY : Integer;
begin
  Show;            { Element darstellen, das verschoben werden soll}
  FigureX := GetX; { Anfangsposition des Elements ermitteln }
  FigureY := GetY;
  { Dies ist nun die Schleife zum Verschieben des Grafikelements. }
  while GetDelta(DeltaX, DeltaY) do
  begin      { Verschiebung auf Element in X,Y anwenden }
    FigureX := FigureX + (DeltaX * DragBy);
    FigureY := FigureY + (DeltaY * DragBy);
    MoveTo(FigureX, FigureY);               { Element verschieben }
  end;
end;

{-------------------------------------------------------------------}
{ Implementierung der Methoden von Circle                           }
{-------------------------------------------------------------------}

constructor Circle.Init(InitX, InitY : Integer;
                        InitRadius : Integer);
begin
  Point.Init(InitX, InitY);
  Radius := InitRadius;
end;
```

```
procedure Circle.Show;
begin
  Visible := True;
  Graph.Circle(X, Y, Radius);
end;

procedure Circle.Hide;
var
  TempColor : Word;
begin
  TempColor := Graph.GetColor;
  Graph.SetColor(GetBkColor);
  Visible := False;
  Graph.Circle(X, Y, Radius);
  Graph.SetColor(TempColor);
end;

procedure Circle.Expand(ExpandBy : Integer);
begin
  Hide;
  Radius := Radius + ExpandBy;
  if Radius  0 then Radius := 0;
  Show;
end;

procedure Circle.Contract(ContractBy : Integer);
begin
  Expand(-ContractBy);
end;

{ kein Initialisierungsabschnitt }
end.
```

Nun sollten Sie soweit sein, daß Sie die Funktionalität in Objekten mit Hilfe von Methoden anstelle von Prozeduren und Parameterlisten realisieren. Letztlich gelangen Sie dahin, daß Sie Programme als Aktivitäten, die von Objekten ausgeführt werden können, definieren und nicht mehr als Sammlung von Prozeduraufrufen, die auf passive Daten angewendet werden.

Erweiterbarkeit von Objekten

Das Wichtigste, was man sich zu Toolbox-Units wie FIGURES.PAS merken sollte, ist die Tatsache, daß die Objekte und Methoden, die in dieser Unit definiert sind, an einen Anwender in einer linkbaren ».TPU«-Datei ausgeliefert werden können, ohne daß man den Quellcode mitliefern muß. (Lediglich ein Listing des Interface-Abschnitts der Unit muß man mitgeben.) Mit Hilfe polymorpher Objekte und virtueller Methoden, können die Anwender von

».TPU«-Dateien immer noch eigene Eigenschaften hinzufügen, um sie an ihre eigenen Bedürfnisse anzupassen.

Diese neuartige Idee, den Programmcode eines anderen zu verwenden und dann weitere Funktionen hinzuzufügen, *ohne dessen Quellcode zu kennen*, wird als *Erweiterbarkeit* bezeichnet. Erweiterbarkeit ist die logische Konsequenz der Vererbung: Es wird alles vererbt, was die Vorfahren auszeichnet und kann dann das hinzufügen, was an neuen Möglichkeiten benötigt wird. Durch die dynamische Bindung werden die neuen Funktionen mit den alten verknüpft, so daß die Erweiterung des existierenden Codes nahtlos vonstatten geht, und sich die Einbuße an Leistungsfähigkeit auf den kurzen Weg über die virtuelle Methodentabelle beschränkt.

Das folgende Programm verwendet die Unit *Figures* und erweitert sie um ein neues Grafikobjekt, *Arc*, als Nachkomme von *Circle*. Das Objekt *Arc* hätte viel später geschrieben werden können, nachdem FIGURES.PAS bereits compiliert worden war, und trotzdem kann ein Objekt des Typs *Arc* die vererbten Methoden wie *MoveTo* oder *Drag* verwenden, ohne daß dazu spezielle Vorkehrungen zu treffen wären. Dynamische Bindung und die virtuellen Methoden von *Arc* machen es möglich, daß *Arc*s Methoden *Show* und *Hide* in *Drag* aufgerufen werden können, auch wenn diese Methoden viel später als *Point.Drag* geschrieben wurden:

```
program FigureDemo;    { erweitert FIGURES.PAS um den Datentyp Arc }

uses Crt, DOS, Graph, Figures;

type
  Arc = object(Circle)
    StartAngle, EndAngle : Integer;
    constructor Init(InitX, InitY : Integer;
                     InitRadius : Integer;
                     InitStartAngle, InitEndAngle : Integer);
    procedure Show; virtual;
    procedure Hide; virtual;
  end;

var
  GraphDriver : Integer;
  GraphMode : Integer;
  ErrorCode : Integer;
  AnArc : Arc;
  ACircle : Circle;

{-------------------------------------------------------}
{ Implementierung der Methoden von Arc                  }
{-------------------------------------------------------}

constructor Arc.Init(InitX, InitY : Integer;
                     InitRadius : Integer;
                     InitStartAngle, InitEndAngle : Integer);
begin
  Circle.Init(InitX, InitY, InitRadius);
```

```
  StartAngle := InitStartAngle;
  EndAngle := InitEndAngle;
end;

procedure Arc.Show;
begin
  Visible := True;
  Graph.Arc(X, Y, StartAngle, EndAngle, Radius);
end;

procedure Arc.Hide;
var
  TempColor : Word;
begin
  TempColor := Graph.GetColor;
  Graph.SetColor(GetBkColor);
  Visible := False;
  { Bogen in der Hintergrundfarbe zeichnen, um ihn zu l·schen }
  Graph.Arc(X, Y, StartAngle, EndAngle, Radius);
  SetColor(TempColor);
end;

{--------------------------------------------------------}
{ Hauptprogramm                                          }
{--------------------------------------------------------}

begin
  Graphdriver := Detect;  { BGI bestimmt, welcher Adapter
                            vorhanden ist }
  InitGraph(GraphDriver, GraphMode, '');
  if GraphResult  GrOK then
  begin
    WriteLn('>Abbruch wegen Grafikfehler:',
            GraphErrorMsg(GraphDriver));
    Halt(1);
  end;
{ Alle Nachkommen des Typs Point enthalten virtuelle Methoden und }
{ *m¡ssen* vor der Verwendung mit einem Konstruktoraufruf }
{ initialisiert werden. }
  ACircle.Init(151, 82,      { X,Y am Anfang bei 151,82 }
               50);          { Radius am Anfang bei 50 Pixel }
  AnArc.Init(151, 82,        { X,Y am Anfang bei 151,82 }
             25, 0, 90);     { Radius am Anfang bei 25 Pixel }
                             { Anfangswinkel: 0, Endwinkel bei 90 }
  { Ersetzen Sie AnArc durch ACircle, um einen Kreis anstelle eines}
  { Kreisbogens zu verschieben. Geben Sie Return ein, um mit dem }
  { Verschieben aufzuh·ren und das Programm zu beenden. }
  AnArc.Drag(5); { Parameter gibt die Anzahl der Pixel
                   f¡r die Verschiebung an }
```

```
  CloseGraph;
end.
```

Statische oder virtuelle Methoden

Im allgemeinen sollten Sie Methoden virtuell anlegen. Verwenden Sie statische Methoden nur dann, wenn Sie die Ausführungsgeschwindigkeit und den Speicherplatzbedarf Ihres Programms optimieren wollen. Der Nachteil liegt, wie Sie gesehen haben, in der fehlenden Erweiterbarkeit.

Nehmen wir einmal an, Sie deklarieren ein Objekt mit der Bezeichnung *Ancestor*, und in *Ancestor* deklarieren Sie eine Methode mit dem Namen *Action*. Wie sollen Sie nun entscheiden, ob *Action* statisch oder virtuell angelegt werden soll? Als Faustregel gilt: Legen Sie *Action* virtuell an, wenn in Zukunft die Möglichkeit besteht, daß ein Nachkomme von *Ancestor* die Methode *Action* ersetzen und für *Ancestor* dieser zukünftige Code verfügbar sein soll.

Wenden Sie diese Regel nun auf die in diesem Kapitel bereits besprochenen Grafikobjekte an. In diesem Fall ist *Point* der Vorfahre. Sie müssen sich also entscheiden, ob dessen Methoden statisch oder virtuell angelegt werden sollen. Sehen wir uns dazu die Methoden *Show*, *Hide* und *MoveTo* etwas genauer an. Da jedes der Grafikobjekte seine eigenen Algorithmen zum Darstellen und Löschen besitzt, werden *Show* und *Hide* in allen Nachkommen ersetzt. Die Verschiebung eines Grafikobjekts scheint jedoch bei allen Nachkommen gleich zu sein: Zunächst wird *Hide* aufgerufen, um das Objekt zu löschen, anschließend werden die *X*,*Y*-Koordinaten verändert und dann wird *Show* aufgerufen, um die Figur wieder an der neuen Position darzustellen. Da dieser *MoveTo*-Algorithmus auf alle Figuren mit einem einzigen Ursprungspunkt an der *X*,*Y*-Position angewendet werden kann, scheint es durchaus sinnvoll zu sein, *Point.MoveTo* als statische Methode einzusetzen, die an alle Nachkommen von *Point* vererbt wird. *Show* und *Hide* werden hingegen ersetzt und müssen virtuell vorliegen, damit *Point.MoveTo* die Methoden *Show* und *Hide* in seinen Nachkommen aufrufen kann.

Auf der anderen Seite sollten Sie beachten, daß bei jedem Objekt mit einer virtuellen Methode eine VMT für diesen Objekttyp im Datensegment erzeugt wird und jede Instanz mit der VMT verbunden sein muß. Jeder Aufruf einer virtuellen Methode verläuft über die VMT, während statische Methoden direkt aufgerufen werden können. Obwohl die Suche in der VMT sehr schnell vor sich geht, ist der Aufruf einer statischen Methode immer ein wenig schneller als der einer virtuellen. Und wenn gar keine virtuellen Methoden in Ihrem Objekt vorliegen, steht auch keine VMT in Ihrem Datensegment. Was noch wichtiger ist, es existieren auch keine Verbindungen von der VMT zu den einzelnen Objektinstanzen.

Die zusätzliche Ausführungsgeschwindigkeit und der geringere Speicherplatzbedarf von statischen Methoden muß gegenüber der Flexibilität der virtuellen Methoden abgewogen werden: die Erweiterbarkeit von existierenden Programmen, lange nachdem der Code compiliert wurde. Berücksichtigen Sie bitte, daß die Anwender Ihrer Objekte möglicherweise auf

Anwendungsmöglichkeiten kommen, von denen Sie nicht einmal geträumt haben, und darauf kommt es an.

Dynamische Objekte

Alle Beispiele von Objekten, die wir bisher vorgestellt haben, hatten statische Instanzen von Objekten, die in einer **var**-Deklaration standen und im Datensegment oder auf dem Stack angelegt waren (beachten Sie bitte, daß das Wort *statisch* nichts mit statischen Methoden zu tun hat).

```
var
  ACircle : Circle;
```

Objekte können aber auch auf dem Heap angelegt werden und mit Zeigern manipuliert werden, genau wie die nah verwandten Rekordtypen. Turbo Pascal enthält einige mächtige Erweiterungen, um die dynamische Speicherplatzreservierung und das Entfernen von Objekten zu erleichtern und zu beschleunigen. Objekte lassen sich als Zeigerreferenzen mit Hilfe der Prozedur *New* anlegen:

```
var
  PCircle : ^Circle;
New(Circle);
```

Wie bei einem Rekord, reserviert *New* genügend Speicherplatz auf dem Heap, um eine Instanz des Basistyps des Zeigers aufzunehmen, und gibt die Adresse dieses Speicherplatzes im Zeiger zurück.

Wenn das dynamische Objekt virtuelle Methoden enthält, dann muß es durch einen Aufruf des Konstruktors initialisiert werden, bevor andere Aufrufe dieser Methoden erfolgen:

```
PCircle^.Init(600, 100, 30);
```

Die Aufrufe der Methoden erfolgen wie üblich mit Hilfe des Zeigernamens und dem Referenzierungssymbol ^(Caret) anstelle der Instanzenbezeichnung, die man beim entsprechenden statischen Objekt verwenden würde:

```
OldXPosition := PCircle^.GetX;
```

Speicherplatzreservierung und Initialisierung mit New

In Turbo Pascal wurde die Syntax von *New* zu einem kompakteren und einfacheren Hilfsmittel erweitert, um die Speicherplatzreservierung und die Initialisierung des Objekts in einem Schritt durchzuführen. *New* kann nun mit zwei Parametern aufgerufen werden: der Zeigername ist der erste, und der Aufruf des Konstruktors ist der zweite Parameter:

```
New(PCircle, Init(600, 100, 30));
```

Wenn Sie diese erweiterte Syntax für *New* verwenden, dann führt eigentlich der Konstruktor *Init* die dynamische Speicherplatzreservierung durch, und zwar in einem speziellen Code-Abschnitt vor dem Einsprungspunkt, der als Teil des Konstruktors während der Compilierung erzeugt wird. Der Instanzenname darf nicht vor *Init* stehen, da zum Zeitpunkt, da *New* aufgerufen wird, die Instanz, die mit *Init* initialisiert wird, noch gar nicht vorhanden ist. Der Compiler erkennt die richtige *Init*-Methode am Typ des Zeigers, der als erster Parameter übergeben wird. *New* wurde auch noch in anderer Hinsicht erweitert, damit man die Prozedur auch als Funktion verwenden kann, die einen Zeiger zurückliefert. Der Parameter, der an *New* übergeben wird, ist der Typ des Zeigers auf das Objekt und nicht auf die Zeigervariable selbst:

```
type
  ArcPtr = ^Arc;
var
  PArc : ArcPtr;
Parc := New(ArcPtr);
```

Beachten Sie bitte, daß die Funktionserweiterung von *New* für *alle* Datentypen und nicht nur für Objekte gilt:

```
type
  CharPtr = ^Char;      { Char ist kein Objekttyp... }
var
  PChar : CharPtr;
PChar := New(CharPtr);
```

Die Funktionsausführung von *New* kann wie die Prozedur mit einem Konstruktor als zweitem Parameter verwendet werden:

```
PArc := New(ArcPtr, Init(600, 100, 25, 0, 90));
```

Parallel dazu wurde die Prozedur *Dispose* erweitert, wie in den nachfolgenden Abschnitten erläutert wird. Ebenfalls neu eingefügt wurde die Standardprozedur *Fail*, die die Fehlerbehandlung in Konstruktoren erleichtert.

Entfernen von dynamischen Objekten

Wie die traditionellen Pascal-Rekords können auch die Objekte, die auf dem Heap angelegt wurden, mit der Prozedur *Dispose* wieder entfernt werden, wenn sie nicht mehr gebraucht werden:

```
Dispose(PCircle);
```

Es kann jedoch manchmal etwas mehr bedeuten, als bloß ein dynamisches Objekt, das man nicht mehr benötigt, freizugeben. Ein Objekt kann Zeiger auf dynamische Strukturen oder Objekte enthalten, die in einer bestimmten Reihenfolge freigegeben oder »aufgeräumt« werden müssen, insbesondere

wenn es sich um sehr ausgefeilte dynamische Datenstrukturen handelt. Was auch immer getan werden muß, um ein dynamisches Objekt ordentlich aufzuräumen: Es sollte in einer einzigen Methode zusammengefaßt werden, damit das Objekt mit einem einzigen Aufruf dieser Methode entfernt werden kann.

```
MyComplexObject.Done;
```

Die Methode *Done* sollte alle Einzelschritte zum Aufräumen des Objekts und aller darin verschachtelten Datenstrukturen und Objekte umfasssen.

Es ist zulässig und oft sogar ganz praktisch, mehrere Methoden zum Aufräumen eines bestimmten Objekts zu definieren. Komplizierte Objekte müssen möglicherweise mit verschiedenen Methoden aufgeräumt werden, je nachdem, wie sie angelegt oder eingesetzt wurden und in welchem Zustand sich das Objekt beim Aufräumen befand.

Destruktoren

Turbo Pascal enthält eine spezielle Art von Methode, die als *Destruktor* bezeichnet wird, um dynamisch angelegte Objekte aufzuräumen und zu entfernen. Ein Destruktor vereinigt den Vorgang der Freigabe des Heap-Speicherplatzes mit allen anderen Aufräumvorgängen, die für einen bestimmten Objekttyp erforderlich sind. Wie bei allen Methoden können mehrere Destruktoren für ein einzelnes Objekt definiert werden.

Ein Destruktor wird zusammen mit all den anderen Methoden des Objekts in der Defintion des Objekttyps angegeben;

```
Point = object(Location)
  Visible : Boolean;
  Next : PointPtr;
  constructor Init(InitX, InitY : Integer);
  destructor Done; virtual;
  procedure Show; virtual;
  procedure Hide; virtual;
  function IsVisible : Boolean;
  procedure MoveTo(NewX, NewY : Integer);
  procedure Drag(DragBy : Integer); virtual;
end;
```

Destruktoren können vererbt werden, sie können entweder statisch oder dynamisch angelegt werden. Da für verschiedene Objekte normalerweise unterschiedliche Abschlußarbeiten erforderlich sind, empfehlen wir Ihnen, Destruktoren *immer* virtuell zu definieren, damit immer der richtige Destruktor für den jeweiligen Objekttyp ausgeführt wird.

Beachten Sie bitte, daß das reservierte Wort **destructor** nicht in allen Aufräummethoden erforderlich ist, auch wenn die Definition des Objekttyps virtuelle Methoden enthält. Destruktoren wirken sich eigentlich nur auf dynamisch reservierte Objekte aus. Beim Aufräumen eines dynamisch reser-

vierten Objekts führt der Destruktor eine spezielle Aufgabe aus: Er garantiert, daß immer die korrekte Anzahl an Bytes auf dem Heap freigegeben wird. Es macht jedoch nichts aus, wenn man Destruktoren zusammen mit statisch angelegten Objekten verwendet. Wenn Sie als Objekttyp keinen Destruktor angeben, dann verhindern Sie im Grunde, daß das entsprechende Objekt die Vorteile der dynamischen Speicherplatzverwaltung unter Turbo Pascal voll ausnützen kann.

Destruktoren werden besonders wichtig, wenn polymorphe Objekte aufgeräumt werden müssen und der auf dem Heap reservierte Speicherplatz wieder freigegeben werden muß. Ein polymorphes Objekt ist ein Objekt, das über die erweiterten Typkompatibilitätsregeln von Turbo Pascal zugewiesen wurde. Im laufenden Beispiel mit den Grafikobjekten ist eine Instanz des Objekts *Circle*, die einer Variable des Typs *Point* zugewiesen wird, ein gutes Beispiel für ein polymorphes Objekt. Diese Regeln gelten auch für Zeiger auf Objekte. Ein Zeiger auf *Circle* kann einem beliebigen Zeiger des Typs *Point* zugewiesen werden, und der vom Zeiger referenzierte Speicherbereich ist ebenfalls ein polymorphes Objekt.

Der Ausdruck *polymorph* ist deshalb angebracht, weil das Programm, in dem das Objekt verwendet wird, zum Zeitpunkt der Compilierung nicht genau wissen kann, welcher Objekttyp am Ende gemeint ist. Es ist lediglich bekannt, daß das Objekt ein Nachkomme in der Objekthierarchie des angegebenen Typs ist.

Die Größe der Objekttypen unterscheidet sich natürlich. Wenn also ein polymorphes Objekt vom Heap wieder entfernt werden soll, wie soll *Dispose* dann entscheiden, wieviele Bytes an Speicherplatz vom Heap wieder freigegeben werden sollen? Zum Zeitpunkt der Compilierung kann jedenfalls über die Größe eines polymorphen Objekts keine Aussage getroffen werden.

Der Destruktor löst dieses Rätsel, indem er sich dorthin wendet, wo die Informationen *gespeichert sind*: an die VMT der Instanz dieser Variablen. In jeder VMT eines Objekts steht die Größe des Objekttyps in Bytes. Die VMT eines Objekts erhält man über den unsichtbaren Parameter *Self*, der an die Methode bei jedem ihrer Aufrufe übergeben wird. Ein Destruktor ist lediglich eine spezielle Art von Methode und erhält daher auch eine Kopie von *Self* auf dem Stack, wenn er von einem Objekt aus aufgerufen wird. Obwohl also ein Objekt *zum Zeitpunkt der Compilierung* polymorph erscheinen mag, ist es zur Laufzeit dank dynamischer Bindung niemals polymorph.

Um die Speicherplatzfreigabe über dynamische Bindung auszuführen, muß der Destruktor als Teil der erweiterten Syntax der *Dispose*-Prozedur aufgerufen werden:

```
Dispose(PPoint,Done);
```

(Der Aufruf eines Destruktors außerhalb eines *Dispose*-Aufrufs führt nie zu einer Speicherplatzfreigabe.) Hier passiert folgendes: Der Destruktor, auf den *PPoint* zeigt, wird wie eine normale Methode aufgerufen. Als letzte Aufgabe aber sucht sich der Destruktor die Größe der Instanz in der VMT heraus und übergibt diese Größe an *Dispose*. *Dispose* schließt die Freigabe damit ab, daß es die richtige Anzahl an Bytes vom Speicherplatz des Heaps freigibt, die vormals zu *PPoint^* gehörten. Die Anzahl der freigegebenen Bytes ist immer

korrekt. Es spielt keine Rolle, ob *PPoint* auf eine Instanz von *Point* oder auf einen Nachkommen von *Point*, wie zum Beispiel *Circle* oder *Arc* zeigt.

Beachten Sie bitte, daß die Destruktormethode selbst leer sein kann und trotzdem noch diese Aufgabe ausführt:

```
destructor AnObject.Done;
begin
end;
```

Was in diesem Destruktor an nützlicher Arbeit ausgeführt wird, steht nicht im Anweisungsblock der Methode, sondern im nachgeschalteten Epilog, der vom Compiler wegen des reservierten Worts **destructor** erzeugt wird. In dieser Hinsicht gleicht er einer Unit, die nichts exportiert, aber eine »unsichtbare« Aufgabe im Initialisierungsabschnitt vor dem Programmstart ausführt. Die Aktion findet hinter dem Vorhang statt.

Ein Beispiel für die dynamische Reservierung von Objekten

Das letzte Beispielprogramm liefert eine praktische Anwendung für den Einsatz von Objekten, die auf dem Heap angelegt werden, zusammen mit dem Einsatz von Destruktoren für das Entfernen von Objekten vom Heap. Das Programm zeigt, wie eine verkettete Liste von Grafikobjekten auf dem Heap erzeugt werden kann und mit Hilfe von Destruktoraufrufen wieder entfernt wird, wenn sie nicht mehr gebraucht wird.

Der Aufbau einer verketteten Liste von Objekten erfordert, daß jedes Objekt einen Zeiger auf das nächste Objekt in der Liste enthält. Der Typ *Point* enthält keinen solchen Zeiger. Am einfachsten wäre es, wenn man einen Zeiger in *Point* hinzufügt, und damit sofort sicherstellt, daß alle Nachkommen von *Point* diesen Zeiger erben. Um jedoch etwas in *Point* hinzuzufügen, braucht man den Quellcode von *Point*, und wie bereits erwähnt wurde, besteht ein Vorteil der objektorientierten Programmierung in der Möglichkeit, vorhandene Objekte zu erweitern, ohne daß man sie erneut compilieren muß.

Bei der Lösung, in der gefordert wird, daß *Point* nicht verändert werden darf, muß ein neuer Objekttyp erzeugt werden, der nicht von *Point* abstammt. Der Typ *List* ist ein sehr einfaches Objekt, dessen Aufgabe darin besteht, eine Liste von *Point*-Objekten anzuführen. Da *Point* keinen Zeiger auf das nächste Objekt in der Liste enthält, muß ein einfacher Rekordtyp, *Node*, eingeführt werden, der diese Aufgabe übernimmt. *Node* ist sogar noch einfacher als *List*. Der Typ ist kein Objekt, hat keine Methoden und enthält keine Daten, außer einem Zeiger auf den Typ *Point* und einen Zeiger auf den nächsten Knoten in der Liste. *List* enthält eine Methode, mit der neue Grafikobjekte in der verketteten Liste als *Node* aufgenommen werden, indem eine neue Instanz von *Node* am Anfang der Liste über den Zeiger auf die Knoten eingefügt wird. Die Methode *Add* kann einen Zeiger auf ein Objekt vom Typ *Point* übernehmen, anstatt eines Objekts *Point*. Wegen der erweiterten Typkompatibilität von Turbo Pascal können alle Zeiger, die auf einen Nachkommen von *Point* zeigen, ebenfalls als Parameter *Item* an *List.Add* übergeben werden.

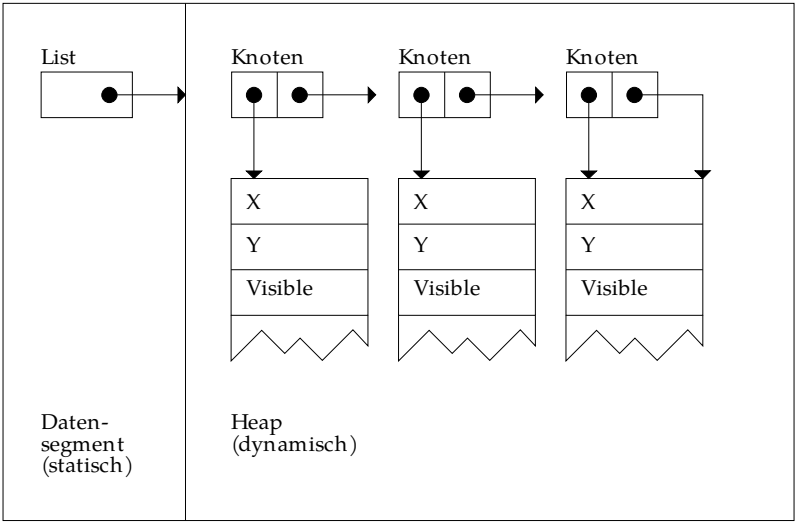


Abbildung 1.2 Aufbau der Datenstrukturen in ListDemo

Das Programm *List.Demo* deklariert eine statische Variable des Typs *List*, *AList*, und erstellt eine verkettete Liste mit drei Knoten. Jeder Knoten zeigt auf ein anderes Grafikobjekt, das entweder *Point* sein kann oder eines seiner Nachkommen ist. Die Anzahl der Bytes auf dem freien Heap wird vor dem Anlegen der dynamischen Objekte ausgegeben und anschließend noch einmal, nachdem sie erzeugt wurden. Schließlich wird die gesamte Struktur zusammen mit den drei *Node*-Rekords und den drei *Point*-Objekten mit Hilfe eines einzigen Destruktoraufrufs des statischen Objekts *List*, *AList*, wieder vom Heap entfernt.

Die Freigabe einer komplizierten Datenstruktur vom Heap

Auf *List.Done* sollte man einen genaueren Blick werfen. Um ein Objekt vom Typ *List* auszulöschen, müssen drei verschiedene Arten von Strukturen entfernt werden: die polymorphen Grafikobjekte in der Liste, die *Node*-Rekords, die die Liste zusammenhalten und (falls sie auf dem Heap angelegt wurde) das Objekt *List*, daß die Liste anführt. Der gesamte Vorgang wird mit einem einzigen Aufruf des Destruktors *AList* ausgeführt:

```
AList.Done;
```

Der Code des Destruktors verdient eine genauere Betrachtung:

```
destructor List.Done;
var
  N : NodePtr;
begin
  while Nodes <> nil do
  begin
```

```
      N := Nodes;
      Dispose(N^.Item, Done);
      Nodes := N^.Next;
      Dispose(N);
    end;
  end;
```

Die Liste wird vom Listenkopf aus mit einem »Hand in Hand« greifenden Algorithmus entfernt, so ähnlich wie beim Aufziehen einer Perlenkette: Zwei Zeiger, der Zeiger *Nodes* in *AList* und ein Arbeitszeiger *N*, wechseln sich beim Zugriff auf die Liste ab, während das erste Element aus der Liste entfernt wird. Ein Aufruf von *Dispose* gibt den Speicherplatz für das erste *Point*-Objekt in der Liste (*Item^*) frei. Anschließend wird *Nodes* zum nächsten *Node*-Rekord in der Liste mit der Anweisung *Nodes := N^.Next* versetzt. Der *Node*-Rekord selbst wird entfernt und der Vorgang wird dann solange wiederholt, bis die gesamte Liste gelöscht ist.

Im Destruktor *Done* kommt es vor allem darauf an, wie die *Point*-Objekte aus der Liste entfernt werden:

```
Dispose(N^.Item,Done);
```

In diesem Fall ist *N^.Item* das erste *Point*-Objekt in der Liste und die Methode *Done* ist sein Destruktor. Bitte merken Sie sich, daß der eigentliche Typ von *N^.Item* nicht unbedingt *Point* sein muß, sondern auch ein Nachkomme von *Point* sein kann. Das Objekt, das entfernt wird, ist ein polymorphes Objekt, und deshalb können keine Annahmen über seine tatsächliche Größe oder den exakten Typ zum Zeitpunkt der Compilierung gemacht werden. In dem früheren Aufruf von *Dispose* ermittelte der »unsichtbare« Abschlußcode in *Done* die Größe der Objektinstanz in der VMT, nachdem alle Anweisungen von *Done* ausgeführt waren. *Done* übergibt diesen Wert an *Dispose*, das wiederum genau den entsprechenden Speicherplatz am Heap freigibt, den das polymorphe Objekt tatsächlich belegt hat.

Prägen Sie sich bitte ein, daß polymorphe Objekte auf diese Weise entfernt werden müssen. Es muß ein Destruktoraufruf an *Dispose* übergeben werden, damit der reservierte Heap-Speicherplatz zuverlässig freigegeben wird.

Im Beispielprogramm ist *AList* als eine statische Variable im Datensegment deklariert. *AList* hätte genausogut selbst auf dem Heap angelegt und mit der Wirklichkeit über einen Zeiger des Typs *ListPtr* verbunden werden können. Wenn der Listenkopf ebenfalls ein dynamisches Objekt gewesen wäre, dann hätte man die Freigabe der Struktur mit einem Destruktoraufruf innerhalb von *Dispose* durchführen können:

```
var
  PList : ListPtr;
...
Dispose(PList,Done);
```

In diesem Fall ruft *Dispose* die Destruktormethode *Done* auf, um die Struktur vom Heap zu entfernen. Anschließend entfernt *Dispose*, wenn *Done*

abgeschlossen ist, den Speicherplatz auf den sich *PList* bezog, und entfernt gleichzeitig den Listenkopf vom Heap.

Das folgende Programm verwendet dieselbe Unit FIGURES.PAS, die wir bereits vorgestellt haben. Es implementiert einen Typ *Arc* als einen Nachkommen von *Point*, erzeugt das Objekt *List*, das als Kopf einer Liste von drei polymorphen Objekten dient, die mit *Point* kompatibel sind, und entfernt anschließend die gesamte dynamische Datenstruktur mit einem einzigen Destruktoraufruf von *AList.Done*.

```
program ListDemo;    { dynamische Objekte & Destruktoren }

uses Graph, Figures;

type
  ArcPtr = ^Arc;
  Arc = object(Circle)
    StartAngle, EndAngle : Integer;
    constructor Init(InitX, InitY : Integer;
                     InitRadius : Integer;
                     InitStartAngle, InitEndAngle : Integer);
    procedure Show; virtual;
    procedure Hide; virtual;
  end;
  NodePtr = ^Node;
  Node = record
    Item : PointPtr;
    Next : NodePtr;
  end;
  ListPtr = ^List;
  List = object
    Nodes : NodePtr;
    constructor Init;
    destructor Done; virtual;
    procedure Add(Item : PointPtr);
    procedure Report;
  end;

var
  GraphDriver : Integer;
  GraphMode : Integer;
  Temp : String;
  AList : List;
  PArc : ArcPtr;
  PCircle : CirclePtr;
  RootNode : NodePtr;

{-------------------------------------------------------}
{ Prozeduren, die keine Methoden sind                   }
{-------------------------------------------------------}

procedure OutTextLn (TheText : String);
```

```
begin
  OutText(TheText);
  MoveTo(0, GetY + 12);
end;

procedure HeapStatus(StatusMessage : String);
begin
  Str(MemAvail : 6, Temp);
  OutTextLn(StatusMessage + Temp);
end;

{-------------------------------------------------------}
{ Implementierung der Methoden von Arc                  }
{-------------------------------------------------------}

constructor Arc.Init(InitX, InitY : Integer;
                     InitRadius : Integer;
                     InitStartAngle, InitEndAngle : Integer);
begin
  Circle.Init(InitX, InitY, InitRadius);
  StartAngle := InitStartAngle;
  EndAngle := InitEndAngle;
end;

procedure Arc.Show;
begin
  Visible := True;
  Graph.Arc(X, Y, StartAngle, EndAngle, Radius);
end;
procedure Arc.Hide;
var
  TempColor : Word;
begin
  TempColor := Graph.GetColor;
  Graph.SetColor(GetBkColor);
  Visible := False;
  Graph.Arc(X, Y, StartAngle, EndAngle, Radius);
  SetColor(TempColor);
end;

{-------------------------------------------------------}
{ Implementierung der Methoden von List                 }
{-------------------------------------------------------}

constructor List.Init;
begin
  Nodes := nil;
end;

destructor List.Done;
var
```

```
  N : NodePtr;
begin
  while Nodes <> nil do
  begin
    N := Nodes;
    Dispose(N^.Item, Done);
    Nodes := N^.Next;
    Dispose(N);
  end;
end;

procedure List.Add(Item :PointPtr);
var
  N : NodePtr;
begin
  New(N);
  N^.Item := Item;
  N^.Next := Nodes;
  Nodes := N;
end;

procedure List.Report;
var
  Current : NodePtr;
begin
  Current := Nodes;
  while Current <> nil do
  begin
    Str(Current^.Item^.GetX : 3, Temp);
    OutTextLn('X = '+Temp);
    Str(Current^.Item^.GetY : 3, Temp);
    OutTextLn('Y = '+Temp);
    Current := Current^.Next;
  end;
end;

{-------------------------------------------------------}
{ Hauptprogramm                                         }
{-------------------------------------------------------}

begin
  Graphdriver := Detect; { BGI bestimmt den vorhandenen Adapter }
  InitGraph(GraphDriver, GraphMode, '');
  if GraphResult  GrOK then
  begin
    WriteLn('>Abbruch wegen Grafikfehler:',
            GraphErrorMsg(GraphDriver));
    Halt(1);
  end;
  HeapStatus('Heap-Status vor dem Anlegen der Liste:       ');
```

```
  { Liste erzeugen }
  AList.Init;
  AList.Add(New(ArcPtr, Init(151, 82, 25, 200, 330)));
  AList.Add(New(CirclePtr, Init(400, 100, 40)));
  AList.Add(New(CirclePtr, Init(305, 136, 5)));
  { Liste durchwandern und X,Y der Elemente in der Liste ausgeben }
  AList.Report;
  HeapStatus('Heap-Status nach dem Anlegen der Liste :     ');
  { gesamte Liste mit einem Destruktoraufruf entfernen }
  AList.Done;
  HeapStatus('Heap-Status nach dem Entfernen der Liste :   ');
  OutTextLn('Dr¡cken Sie Return, um das Programm zu beenden');
  ReadLn;
  CloseGraph;
end.
```

Wie soll's weitergehen?

Wie in fast allen Gebieten der Computerprogrammierung werden Sie das Konzept der objektorientierten Programmierung nicht dadurch besser verstehen, daß Sie etwas darüber lesen, sondern nur durch Ausprobieren. Die meisten Leute, die sich zum ersten Mal mit objektorientierter Programmierung beschäftigen, hört man mit einem Stoßseufzer murmeln: »Ich kapier das nicht«. Das »Aha« kommt erst später, wenn sich bei der Definition der ersten eigenen Objekte das Gesamtkonzept in dem Moment zusammenfügt, den wir als den Moment der Erleuchtung bezeichnen. Wie das Gesicht der Frau aus einer Rorschach-Zeichnung hervortritt, so tritt das, was vorher so unverständlich war, klar und deutlich vor Augen, und von diesem Moment an gibt es keine Probleme mehr. Für Ihr erstes objektorientiertes Projekt sollten Sie am besten die Unit FIGURES.PAS verwenden (sie liegt auf Diskette vor) und erweitern. Punkte, Kreise und Kreisbögen sind bei weitem noch nicht genug. Erzeugen Sie Objekte für Linien, Rechtecke und Quadrate. Wenn Sie noch etwas ehrgeiziger sind, können Sie ein Kreisdiagramm mit Hilfe einer verketteten Liste von einzelnen Kreissegmenten erzeugen.

Eine etwas höhere Herausforderung ist die Implementierung von Objekten mit relativen Positionen. Eine relative Position ist ein Abstand von einem bestimmten Ausgangspunkt aus, der als positive oder negative Differenz ausgedrückt wird. Ein Punkt mit den relativen Koordianten -17,42 befindet sich 17 Pixel links vom Ausgangspunkt entfernt und 42 Pixel unterhalb dieses Ausgangspunkts. Die relativen Koordinaten sind notwendig, um einzelne Figuren zu einem größeren Grafikobjekt zusammenzufassen, da eine Kombination aus Grafikobjekten nicht immer über die Ausgangspunkte der einzelnen Objekte verbunden werden kann. Es ist besser, wenn man zusätzlich zum Punkt *X*, *Y* ein Feld *RX* und ein Feld *RY* definiert und die sich daraus ergebende Endposition eines Objekts auf dem Bildschirm als Summe des Ursprungs und

der relativen Koordinaten betrachtet. Sobald Sie Ihren »Aha-Effekt« haben, sollten Sie damit beginnen, die Konzepte der objektorientierten Programmierung zur Lösung Ihrer alltäglichen Aufgaben einzusetzen. Nehmen Sie ein paar bereits vorhandene Hilfsprogramme, die Sie jeden Tag verwenden und überdenken Sie diese Programme aus der Sicht der objektorientierten Programmierung. Werfen Sie einen weiteren Blick auf den Mischmasch Ihrer Prozedurbibliotheken und versuchen Sie die Objekte darin zu erkennen – anschließend schreiben Sie diese Prozeduren nochmal mit Objekten. Sie werden erkennen, daß Bibliotheken mit Objekten viel leichter in zukünftigen Projekten wiederverwendet werden können als Ihre bisherigen Bibliotheken. Nur sehr wenig von dem, was Sie in die Programmierung der ersten Version investiert haben, wird sich als unbrauchbar erweisen. Sie werden kaum noch ein Objekt von Grund auf neu schreiben müssen. Wenn etwas fehlt, werden Sie es erweitern. Aber wenn's funktioniert, gibt es keinen Grund, etwas davon wegzuwerfen, was bereits vorhanden ist.

Zusammenfassung

Die objektorientierte Programmierung ist eine direkte Antwort auf die Komplexität moderner Anwendungen – eine Komplexität, die viele Programmierer nahezu verzweifeln läßt. Vererbung und Kapselung sind äußerst effektive Mittel, um diese Komplexität in den Griff zu bekommen. (Es ist derselbe Unterschied, wie zwischen 10000 Insekten in einer Taxonomie-Karte und 10000 Insekten, die einem um die Ohren schwirren.) Weitaus mehr als die strukturierte Programmierung erzwingt die objektorientierte Programmierung eine durchdachte Ordnung in den Software-Strukturen, die wie eine Bestimmungskarte eine Ordnung erfordert, ohne eine Beschränkung aufzuerlegen.

Wenn Sie zu diesem Versprechen noch die Erweiterbarkeit und Wiederverwendbarkeit von bestehenden Programmen hinzunehmen, dann klingt das Ganze fast schon zu schön, um wahr zu sein. Kann gar nicht sein, meinen Sie?

Aber Achtung! Sie haben es mit Turbo Pascal zu tun. Nichts ist unmöglich.

2. Das Rad neu erfinden?

Ihr letztes Programm: wieviel davon war der unvermeidlichen Bürokratie gewidmet? Wieviel Programmcode mußten Sie schreiben, um Menüs zu verwalten, Eingaben des Benutzers entgegenzunehmen, Fehlermeldungen auszugeben usw. usw.? Und wie groß war dagegen der produktive Teil ihres Programmes, der Teil also, der die Aufgaben erledigt, um deretwillen das Programm eigentlich geschrieben wurde? Es ist nun einmal so, daß jedes größere Programm, ob es Daten verwaltet oder Text verarbeitet, eine Menge solcher bürokratischer Arbeiten erledigen muß. Und daher werden Eingaberoutinen, Menüverwaltungen, Fensterroutinen immer wieder neu programmiert.

Nein – erben!

Sie kennen die Warnung: Verschwenden Sie Ihre Zeit nicht damit, das Rad neu zu erfinden. Und mit Turbo Vision wird endlich wahr, was die Propheten der objektorientierten Programmierung immer wieder versprochen haben: Es gibt etwas zu erben.

Was? Nun, kurz gesagt bildet Turbo Vision das Gerüst für ein ereignisgesteuertes, fensterorientiertes Programm. Kein produktiver Teil, nur Bürokratie. Die produktiven Teile Ihrer Programme müssen Sie selbst entwickeln, indem Sie Turbo Vision erweitern. Turbo Vision liefert Ihnen das grundlegende Objekt für Ihre Anwendungen: *TApplication*. Sie definieren Ihre Anwendung als ein davon abgeleitetes Objekt – nennen wir es *MeineApplikation*. Und diesem Objekt fügen Sie alle jene Teile hinzu, die notwendig sind, die beabsichtigten Aufgaben zu erledigen.

Das wäre eigentlich alles. Aus dieser zugegeben abstrakten Sicht betrachtet, könnte der Anweisungsteil Ihres Programms so aussehen:

```
begin
  MeineApplikation.Init;
  MeineApplikation.Run;
  MeineApplikation.Done;
end.
```

Eine neue Vision der Programmentwicklung

Sicherlich haben auch Sie schon Bibliotheken vordefinierter Prozeduren und Funktionen benutzt. Solche Routinen-Sammlungen werden von vielen Seiten angeboten, und viele davon bieten Menü-Routinen, Verwaltung von Bildschirmfenstern, Mausunterstützung. Auf den ersten Blick scheint Turbo Vision nur eine weitere dieser Routinen-Bibliotheken zu sein. Aber unter dieser oberflächlichen Ähnlichkeit verbirgt sich ein radikaler Unterschied, den Sie kennen müssen, wenn Sie das Konzept von Turbo Vision verstehen wollen.

Mit Turbo Vision betreten Sie das Reich der Objekte. Wenn in der traditionellen strukturierten Programmierung ein Programmteil, der zum Beispiel Menüs zu verwalten hatte, Ihren Anforderungen nicht entsprach, dann änderten Sie den Quellcode. Dieses Vorgehen hat jedoch oft schwerwiegende Konsequenzen. Bewährten und erprobten Programmcode zu ändern heißt, eine Schneise schlagen für neue Fehler, die auszumerzen oft mehr Zeit kostet als Sie ursprünglich für die Änderungen veranschlagt hatten.

Verwenden Sie Turbo Vision, und Sie müssen nie mehr im Quellcode herumstochern. Denn Sie ändern Turbo Vision, indem Sie Turbo Vision erweitern. Das grundlegende Objekt *TApplication* in APP.TPU bleibt davon unberührt. Sie erweitern es, indem Sie von ihm abgeleitete Objekte definieren. Sie ändern es, indem Sie die vorhandenen Methoden durch neue Methoden überladen, Methoden, die Sie für diese abgeleiteten Objekte schreiben.

Turbo Vision stellt eine hierarchische Ordnung von Objekten dar, keine Werkzeugkiste voll von Instrumenten, die Sie unabhängig voneinander benutzen können. Bei Turbo Vision gilt: alles oder nichts. Denn kein Teil von Turbo Vision arbeitet unabhängig von den anderen Teilen. Versuchen Sie nicht, zum Beispiel nur jene Routinen herauszuziehen, die für die Unterstützung der Maus verantwortlich sind. Letzten Endes würde ein solches Vorgehen mehr Arbeit kosten als nötig wäre, diese Routinen selbst neu zu schreiben.

Diese beiden Eigenschaften sind grundlegend für die Ideen, die hinter der Entwicklung von Turbo Vision standen. Das heißt für Sie: Wenn Sie Turbo Vision benutzen, müssen Sie die Techniken objektorientierten Programmierens anwenden, Sie müssen Ihre Anwendung auf jene grundlegenden Objekttypen aufbauen, die Turbo Vision zur Verfügung stellt, und Sie müssen Turbo Vision die Erledigung jener Aufgaben überlassen, für die Turbo Vision zuständig ist. Sie sind vielleicht der Meinung, dies schränke Sie ein, und in einem gewissen Sinn stimmt das auch. Aber Sie können jederzeit eigene Routinen schreiben, die genau jene Aufgaben erledigen, die auch Turbo Vision erledigen könnte. Wir jedenfalls haben Turbo Vision entwickelt, um Ihnen eine Menge eigentlich unnötiger und sich doch immer wiederholender Arbeit zu ersparen. Wir wollten Ihnen ein erprobtes Fundament zur Verfügung stellen, dem Sie vertrauen können.

Die Elemente eines Turbo-Vision-Programms

Bevor wir uns genauer ansehen, wie ein Programm arbeitet, das auf Turbo Vision aufbaut, betrachten wir die grundlegendsten Elemente von Turbo Vision.

Jedes Turbo-Vision-Programm ist eine Versammlung miteinander kooperierender Komponenten, die aus *View*-Objekten, Ereignissen und »stummen« Objekten bestehen.

Views

Als *View* kann jedes Programmelement betrachtet werden, das sich irgendwie auf dem Bildschirm bemerkbar macht. Im Zusammenhang mit Turbo Vision ist alles, was auf dem Bildschirm zu sehen ist, eine *View*. Eingabefelder, Rollbalken, Fensterrahmen, Dialogfenster, all dies sind *Views*. Sie sind rechteckig und möglicherweise nur eine Spalte breit und eine Zeile hoch, können aber auch den gesamten Bildschirm überdecken. Views können miteinander kombiniert werden und so komplexere Darstellungen ermöglichen, wie beispielsweise ein Bildschirmfenster innerhalb dessen der Inhalt einer Textdatei dargestellt wird. Eine solche Kombination wird Gruppe genannt, und die Elemente dieser Gruppe arbeiten so eng zusammen, daß die Gruppe selbst wieder als einzige View betrachtet werden kann. In Kapitel 3, »*View-Objekte*« gehen wir näher auf diese Gruppe von Objekten ein.

Events

Ein Event ist ein Ereignis, auf das Ihr Programm irgendwie reagieren muß. Ein Event kann über die Tastatur oder mittels der Maus, aber auch durch das Programm selbst ausgelöst werden. Turbo Vision reiht solche Events in der Folge, in der sie eintreffen, in eine Warteschlange ein und reicht sie auch in dieser Reihenfolge an einen sogenannten Event-Handler weiter. Das Objekt *TApplication*, auf das jedes Turbo-Vision-Programm aufbaut, definiert einen solchen Event-Handler, genau wie im Prinzip jede andere View. Events, die *TApplication* nicht selbst bedient, werden an die anderen Views, die zu dem Programm gehören, weitergereicht, bis schließlich innerhalb dieser Kette eine zuständige View gefunden ist, deren Event-Handler auf das Ereignis reagiert. (Auf den Mechanismus, der hinter diesem »Weiterreichen« steht, werden wir später genauer eingehen.)

In vielen Anwendungen kann der Benutzer zum Beispiel über [F1] Hilfestellungen abrufen. Sofern nicht jedes View-Objekt einen eigenen Zugang zum Hilfesystem definiert (was innerhalb eines kontextsensitiven Hilfesystems notwendig sein kann), wird ein Druck auf [F1] durch den Event-Handler des Hauptprogramms abgearbeitet. Dagegen müssen »normale« Tasten von dem Event-Handler der gerade aktiven View entgegengenommen und abgearbeitet werden.

Stumme Objekte

sind alle jene Objekte eines Programms, die keine Views sind. Sie führen Berechnungen durch, kommunizieren mit Peripheriegeräten, kurz: bilden den produktiven Teil Ihres Programmes. Sie sind stumm, weil es ihnen nicht gestattet ist, sich auf dem Bildschirm bemerkbar zu machen. Wenn ein solches Objekt zum Beispiel eine Meldung auf dem Bildschirm ausgeben muß, so kann es dies nur tun in Zusammenarbeit mit einer View. Und das ist ein wichtiges Gebot: Nur Views ist es gestattet, auf den Bildschirm zuzugreifen.

Natürlich hindert Sie niemand daran, über die normalen *Write*- bzw. *WriteLn*-Anweisungen Meldungen auf dem Bildschirm auszugeben. Sie sollten es dennoch nicht tun. Denn erstens stören solche an Turbo Vision vorbeigeschmuggelten Meldungen dessen eigene Bildschirmdarstellungen. Zum zweiten weiß Turbo Vision nichts von diesen Informationen, kann sie daher auch nicht berücksichtigen und wird sie wahrscheinlich sehr bald überschreiben. Daher: Nur Turbo Vision hat die Kompetenz, den Bildschirm zu nutzen, und Sie müssen sich seiner Möglichkeiten bedienen, wenn Ihre Objekte Informationen auf dem Bildschirm ausgeben wollen.

Da Turbo Vision in der Absicht entwickelt wurde, ein standardisiertes und verständliches Bildschirm-Design für interaktive Anwenderprogramme zu bieten, werden Ihre Turbo-Vision-Programme sehr bald ein vertrautes »Gesicht« mit einer eingängigen und bekannten Bedienungsweise verbinden. Dieses Design ist dasselbe, das auch die Turbo-Sprachen zeigen, und es basiert auf Jahren der Entwicklung und ausgiebiger Tests. Ein solches wohlvertrautes Äußeres von Programmen ist ein Vorteil gleichermaßen für Sie selbst wie für die Benutzer Ihrer Programme: Gleichgültig wie neu und ungewöhnlich Ihr Programm sein mag bezüglich dessen, was es tut, die Art und Weise es zu benutzen, wird dem Benutzer immer vertraut sein, und er wird sich sehr viel schneller und leichter mit dem Programm zurechtfinden.

Abbildung 2.1 zeigt einige der Objekte, die allen Programmen eigen sind, die auf Turbo Vision aufbauen. Das Desktop stellt sich dar als gerasterte Fläche, die der Rest des Programms als Hintergrund für seine Ausgaben nutzt.

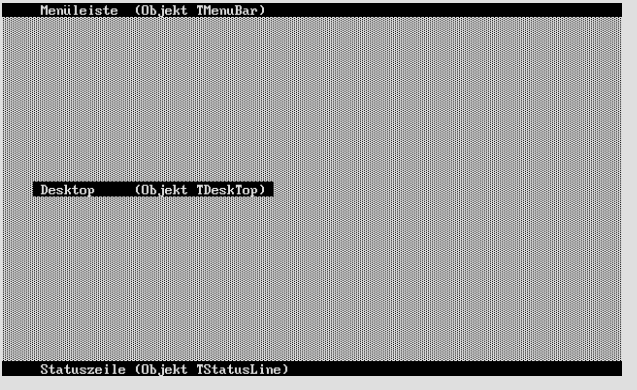


Abbildung 2.1 Einige Objekte auf dem Bildschirm

Wie alles in Turbo Vision ist natürlich auch das Desktop ein Objekt. Begrenzt wird das Desktop nach oben durch eine Menüleiste, nach unten durch die Statuszeile. Wörter in der Menüleiste stehen für weitere Menüs, die aufklappen, wenn Sie mit dem Mauszeiger dieses Wort anklicken oder jene Taste betätigen, die als Tastenkürzel für das betreffende Menü vorgesehen wurde.

Welcher Text in der Statuszeile erscheint, ist Ihnen überlassen. Normalerweise werden hier Meldungen über den aktuellen Zustand des Programms ausgegeben oder aber eine Auflistung der im Moment verfügbaren Kommandos.

Wird ein Pull-Down-Menü geöffnet, gleitet, übereinstimmend mit den Mausbewegungen, ein Leuchtbalken über die Liste der innerhalb des Menüs verfügbaren Optionen. Drücken Sie die linke Maustaste, so wird jene Option ausgewählt, auf der sich der Balken befand, als Sie die Taste betätigten. Normalerweise wird dadurch gleichzeitig ein Befehl an einen anderen Teil des Programms abgesetzt.

Ihr Programm kommuniziert mit dem Benutzer über Bildschirmfenster, die sich auf dem Bildschirm öffnen oder schließen, entsprechend den über die Tastatur oder die Maus eingegebenen Befehlen. Turbo Vision stellt Ihnen ein ausgeklügeltes Fenstersystem zur Verfügung, über das Meldungen ausgegeben und Eingaben angefordert werden können. Der Fensterinhalt kann horizontal und vertikal gerollt werden. Ein solches Fenster verhält sich damit wie ein Guckloch, durch das Sie Ausschnitte auch einer wesentlich größer dimensionierten Darstellung von Daten sehen können. Ein solches Fenster ist rechts oder unten, möglicherweise auch an beiden Seiten begrenzt durch Rollbalken, die einerseits dazu dienen, das Fenster über die Daten zu bewegen, andererseits aber auch die relative Position des im Fenster dargestellten Ausschnitts innerhalb des gesamten Datenbereichs anzugeben.

Verfügbar sind auch sogenannte Aktionsschalter: Farblich hervorgehobene Wörter, die mittels des Mauszeigers »angeklickt« werden. Die dadurch ausgelöste Farbänderung des dargestellten Wortes erweckt den Eindruck, als würde ein wirklicher Schaltknopf niedergedrückt werden. Auch solche Aktionsschalter dienen dazu, anderen Teilen Ihres Programms Befehle zu übermitteln.

Ein einfaches Turbo-Vision-Programm

Traditionellerweise werden die Möglichkeiten einer neuen Programmiersprache oder eines damit vergleichbaren Werkzeugs durch ein Programm der Welt vorgestellt, das diese mit einem herzhaften »Hallo« begrüßt. Ein solches Programm besteht aus nicht mehr Code als notwendig ist, die Zeichenkette »Hallo Welt« auf dem Bildschirm auszugeben, und dann zum Betriebssystem zurückzukehren.

Mit dieser Tradition wollen wir hier nicht brechen, müssen aber doch auf einen grundlegenden Unterschied zwischen dem auf Turbo-Vision aufbauenden Programm und einem klassischen »Hallo-Welt«-Programm aufmerksam

machen. Denn dieses ist ja keineswegs interaktiv, wogegen Turbo Vision gerade die Entwicklung interaktiver Programme unterstützt.

Selbst das einfachste Turbo-Vision-Programm umfaßt weit mehr, als eine zwischen *begin* und *end* eingeschobene *WriteLn*-Anweisung. Das Listing finden Sie in der Datei HELLO.PAS auf den Originaldisketten. Verglichen mit dem klassischen »Hallo-Welt«-Programm erledigt das Turbo-Vision-Pendant weit mehr Aufgaben:

1. Es löscht den Bildschirm, indem es ihn mit Rasterzeichen füllt,

2. es stellt eine Menüleiste oben und eine Statuszeile unten dar,

3. initialisiert einen Event-Handler für die Abarbeitung der von Tastatur oder Maus eingehenden Events,

4. baut »hinter der Bühne« ein Menü auf und verbindet es mit der Menüleiste,

5. verbindet mit diesem Menü ein Dialogfenster, das ebenfalls hinter der Szene aufgebaut wird

6. und wartet schließlich darauf, daß Sie entweder über die Tastatur oder die Maus einen gültigen Befehl eingeben.

Nirgends in dieser Liste taucht der Punkt »Text ausgeben« auf. Natürlich: Text wurde im Hintergrund zur Ausgabe vorbereitet und wartet nun auf den Befehl, der ihm erlaubt, aus diesem Hintergrund heraustreten und auf dem Bildschirm erscheinen zu können. Und dies sollten Sie im Auge behalten, während Sie Turbo Vision kennenlernen: *Das Wesentliche in Turbo Vision ist die Eingabe und die Abarbeitung von Befehlen.* Den produktiven Teil eines Programms müssen Sie schreiben, Turbo Vision seinerseits erledigt die Bürokratie, nimmt Befehle entgegen, überprüft sie auf ihre Gültigkeit und übermittelt sie den für ihre Bearbeitung zuständigen Programmteilen.

Bevor wir auf die Komponenten von HELLO.PAS näher eingehen, laden wir das Programm, kompilieren es, und beobachten, was das Programm im einzelnen tut.

Nach dem Start löscht HELLO den Bildschirm und baut ein Desktop auf,

Abbildung 2.2
Start-Bildschirm von HELLO.PAS

wie in Abbildung 2.2 gezeigt. Keinerlei Fenster sind geöffnet, und die Menüleiste am oberen Rand des Bildschirm zeigt nur einen einzigen Eintrag, den Befehl *Hello*. Beachten Sie, daß das H in einer anderen Farbe als das i dargestellt ist, und daß in der Statuszeile der Hinweis *Alt-X Exit* erscheint.

Das entspricht einer generellen Regel, die bei der Programmierung jeder Benutzerumgebung gilt: *Lassen Sie den Benutzer nicht im Ungewissen darüber, was er als nächstes tun kann.* Der Benutzer von HELLO hat zwei Möglichkeiten: Er kann in der Menüleiste die Option *Hello* anwählen oder das Programm über [Alt][X] verlassen. Und auch dies entspricht gutem Stil: Eröffnen Sie dem Benutzer zwei Wege, einen nach vorn und einen zurück. Denn alles was Sie mit einiger Sicherheit voraussetzen können ist, daß der Anwender Ihrer Programme lesen kann, daß er mit der Tastatur umzugehen versteht, und daß er die Maus zu handhaben weiß.

Das Menü aktivieren

Wählen Sie nun die Option *Hello* in der Menüleiste. Über drei Wege können Sie dies erreichen:

- Bewegen Sie den Mauszeiger auf das Wort Hi und klicken Sie es mit der linken Maustaste an;
- drücken Sie [F10], worauf die Menüleiste aktiviert und *Hello* in anderen Farben dargestellt wird. Darauf drücken Sie [Enter] um *Hello* anzuwählen;
- drücken Sie die Taste [Alt] in Verbindung mit der Taste, die dem hervorgehobenen Zeichen in Hi entspricht, also [H].

In jedem Fall öffnet sich ein Menü unterhalb der Option *Hello*. Diese Funktionsweise ist Ihnen als einem Anwender der Turbo-Sprachen sicherlich vertraut. Denn genauso funktioniert auch die integrierte Entwicklungsumgebung von Turbo Pascal. Sie werden feststellen, daß Turbo Vision nahezu allen Konventionen dieser Entwicklungsumgebung entspricht.

Das nun geöffnete Menü zeigt lediglich zwei Optionen, voneinander getrennt durch eine horizontale Linie. Diese Linie teilt das Menü in zwei Bereiche. HELLO selbst ist ein so einfaches Programm, daß jeder Bereich nur jeweils eine Option zur Wahl anbietet. Jedoch kann ein Menü eine beliebige Zahl von Optionen aufnehmen. Nur die physikalische Ausdehnung des Bildschirms begrenzt deren Zahl.

Auswählen können Sie eine Menüoption sowohl über die Tastatur als auch mit der Maus. Die Cursortasten bewegen den hervorgehobenen Auswahlbalken über die einzelnen Menüpunkte; drücken Sie [Enter], und die Option wird aktiviert, auf der sich dieser Balken gerade befand. Interessanter ist die Auswahl über die Maus: Sie fassen den Auswahlbalken, indem Sie den Mauszeiger über ihm positionieren und dann die linke Maustaste drücken. Solange Sie diese Taste niedergedrückt halten, können Sie mit der Maus den Auswahlbalken über die im Menü enthaltenen Optionen bewegen. Sie wählen eine

Option aus, indem Sie die Maustaste loslassen, während der Balken die Sie interessierende Option überdeckt.

Ein Dialogfenster

Sobald Sie die Option ***G**reeting* auswählen, erscheint ein rechteckiger Ausschnitt, ein Dialogfenster (Abbildung 2.3). Dieses Dialogfenster öffnet sich im Zentrum des Bildschirms; Sie können es jedoch auf dem Bildschirm umherbewegen, indem Sie den Mauszeiger auf die obere Zeile des Rahmens bewegen.

Abbildung 2.3
Das geöffnete Dialogfenster

Drücken Sie dann auf die linke Maustaste. Solange Sie diese Taste niedergedrückt halten, wird das Dialogfenster den Bewegungen der Maus folgen. Lassen Sie die Taste los, und das Fenster wird dort verharren, wo sie sich gerade befand.

Der Rahmen des Dialogfensters wird von zwei Elementen unterbrochen: einem Titel, »Hello, World!« in diesem Fall, und einem Schließfeld nahe der linken oberen Ecke. Klicken Sie dieses Schließfeld an und das Dialogfenster verschwindet wieder. Innerhalb des Dialogfensters findet sich ein kurzer Text: »How are you?«. Das ist ein Beispiel für statischen Text: Er kann gelesen werden, wenn Sie ihn jedoch mit der Maus anklicken, passiert überhaupt nichts. Solch statischer Text dient dazu, Hinweise zu geben und Informationen darzustellen.

Schalter

Weitaus interessanter sind die vier Rechtecke im rechten Teil des Dialogfensters. Sie werden Aktionsschalter genannt und zwar deswegen, weil sie Schaltelementen an elektronischen Geräten ähneln. Jeder dieser Schalter ist mit einer Bezeichnung versehen, die angibt, was passiert, wenn dieser Schaltknopf betätigt wird. Sie »drücken« einen Schalter, indem Sie ihn mit der Maus

anklicken, oder indem Sie ihn aktivieren und dann [Enter] drücken. Betätigen Sie nun einen dieser Schalter mit der Maus: halten Sie die linke Maustaste niedergedrückt, während sich der Mauszeiger auf dem gewählten Schaltelement befindet. Sie werden sehen, daß sich die Beschriftung um eine Spalte nach rechts bewegt, und der Schatten des Schalters verschwindet. Das erweckt den Eindruck, als würde ein Schalter auf dem Bildschirm niedergedrückt. Sobald Sie nun die Maustaste loslassen, wird die angegebene Aktion ausgeführt.

Beachten Sie, daß der »Cancel«-Schalter in anderer Farbe etikettiert ist als die anderen Schalter. Dieser Farbunterschied zeigt an, daß dieses Element das im Augenblick aktuelle ist: Wenn Sie jetzt [Enter] drücken, dann betätigen Sie in Wirklichkeit den »Cancel«-Schalter.

Mittels der Taste [Tab] können Sie einen der anderen Schalter zum aktuellen machen. Sie werden sehen, daß dann dessen Bezeichnung in der unterscheidenden Farbe erscheint. Der Benutzer kann also auch ohne Maus den gewünschten Schalter betätigen, indem er mittels [Tab] das gewählte Element aktiviert und dann über [Enter] oder [Leertaste] »auf den Knopf drückt«.

Den Dialog abschließen

In unserem Fall führen alle Schalter dazu, das Dialogfenster zu schließen und die leere Arbeitsfläche zu restaurieren. Sie können nun erneut das Menü *Hi* und darauf das Dialogfenster öffnen. Wenn Sie aber das Programm verlassen wollen, dann wählen Sie die Option *Exit* in diesem Untermenü oder benutzen einfach die dafür vorgesehene Tastenkombination [Alt][X] . Gleich zweimal wird auf dieses Tastenkürzel hingewiesen: in der Statuszeile am unteren Rand des Bildschirms und zusätzlich im Menü *Hello* selbst.

Solche Informationsredundanz ist keineswegs überflüssig: Machen Sie es den Benutzern Ihrer Programme immer leicht, das Programm auf reguläre Weise zu verlassen. Nur zu leicht neigt ein frustrierter Benutzer, der nicht mehr weiterkommt, dazu, seinen Computer neu zu booten – und hindert damit das Programm an der Ausführung notwendiger Abschlußarbeiten, etwa noch offene Dateien zu schließen.

HELLO.PAS von innen

Wir wissen nun, was HELLO.PAS macht; auf das Wie wollen wir im folgenden eingehen. Allerdings – der weitaus größte Teil von HELLO ist Erbgut, geerbte Eigenschaften der in Turbo Vision definierten Objekte. Tatsächlich ist soviel an HELLO Erbgut, daß den Ablauf des Programmes den Hauch eines Geheimnisses umgibt. Und der Schleier des Geheimnisses lüftet sich auch dann nicht, wenn Sie den Programmlauf mittels des integrierten Debuggers verfolgen, denn schließlich wird Turbo Vision in Form kompilier-

ter Units geliefert. Wenn Sie sich jedoch die Zeit nehmen zu verstehen, was passiert, ist das genaue Wie glücklicherweise nicht mehr so wichtig.

Erinnern Sie sich daran, daß jedes Programm, das auf Turbo Vision aufbaut, eine Sammlung miteinander kooperierender Objekte darstellt. Wenn Sie ein solches Programm verstehen wollen, müssen Sie zunächst dessen grundlegenden Objekte und ihr Zusammenwirken verfolgen. Versuchen Sie dann zu verstehen, wie die Arbeit dieser Basis-Objekte durch andere, weniger umfassende Objekte unterstützt wird. Lesen Sie dazu zuerst die jeweilige Definition des Objekts. Gehen Sie zur Betrachtung der konkreten Implementierung der einzelnen Methoden erst dann über, wenn Sie verstanden haben, was das jeweilige Objekt enthält und in welcher Weise es mit den anderen Objekten zusammenhängt.

Das Objekt TApplication

Das fundamentale Objekt in einem Turbo-Vision-Programm ist das Objekt *TApplication*. Sie werden niemals ein Objekt dieses Typs direkt instantiieren, denn *TApplication* ist ein abstrakter Objekttyp und tut überhaupt nichts. Wozu *TApplication* dann gut ist? Nun, Sie definieren ein davon abgeleitetes Objekt, und dieses abgeleitete Objekt enthält alle nützlichen, produktiven Eigenschaften, die Ihr Programm auszeichnen.

In HELLO nennt sich dieser abgeleitete Objekttyp *THelloApp*:

```
PHelloApp = ^THelloApp;
THelloApp = object(TApplication)
  constructor Init;
  procedure HandleEvent(var Event : TEvent); virtual;
  procedure InitMenuBar; virtual;
  procedure InitStatusLine; virtual;
  procedure GreetingBox;
end;
```

THelloApp enthält natürlich wesentlich mehr als nur die vier hier aufgeführten Methoden, denn ein abgeleitetes Objekt erbt alle Eigenschaften seiner Vorfahren. Die Definition von *THelloApp* legt lediglich fest, wodurch sich dieses neue Objekt von seinem Vorfahren *TApplication* unterscheidet. Alles, was nicht umdefiniert wurde, übernimmt dieses neue Objekt unverändert von *TApplication*.

Näher betrachtet umreißen diese vier Methoden jedoch ein abstraktes Bild des ganzen Programms:

- Der Konstruktor *Init* reserviert Speicher für das Programm, belegt die Variablen mit Anfangswerten, usw. Beachten Sie, daß *THelloApp.Init* seinerseits den Konstruktor seines Vorfahren – *TApplication.Init* – aufruft, der seinerseits die meiste Arbeit bei der Initialisierung des Programms übernimmt.

- Was das Programm genau macht, wird wesentlich bestimmt dadurch, auf welche Ereignisse es reagiert und wie es auf solche Ereignisse reagiert. Daher

muß das Objekt eine *HandleEvent*-Methode definieren. Eine solche Methode ist in *TApplication* schon enthalten; ihre Aufgabe ist es, all jene allgemeinen Ereignisse abzuarbeiten, die in jedem Programm auftreten. Jene Ereignisse jedoch, die spezifisch für Ihr Programm sind, müssen von einem Event-Handler abgearbeitet werden, den Sie definieren.

- Die Methode *InitMenuBar* baut das Menü für das Programm auf und verbindet es mit der Menüzeile. *TApplication* hat eine solche Menüzeile, jedoch keine Menüs. Wenn Sie also Menüs ausgeben wollen (und welch größeres Programm kann schon darauf verzichten), müssen Sie eine Methode entwickeln, die diese Menüs definiert. Sie fragen sich vielleicht, warum die Aufgabe von *InitMenuBar* nicht gleich durch den Konstruktor von *THelloApp* mit erledigt wird. Das wäre natürlich möglich. Aber ein komplexeres Programm als HELLO.PAS wird wahrscheinlich mehrere unterschiedliche Menüs zur Wahl bereitstellen, die auch zu Beginn des Programms schon aktiv sein können. Daher ist es besser, solche Aufgaben separat zu erledigen, und dem Konstruktor nur das aufzutragen, was immer auf die gleiche Weise erledigt werden muß, wenn das Programm gestartet wird.

- Die Methode *GreetingBox* öffnet das Dialogfenster immer dann, wenn die Menüoption ***G**reeting* gewählt wird. Genauer gesagt wird *GreetingBox* von der Methode *HandleEvent* aufgerufen als Antwort auf ein Ereignis, das durch die Auswahl der Menüoption *Greeting* ausgelöst wurde. Natürlich werden komplexere Programme mehrere Methoden definieren, entsprechend einer größeren Anzahl von Menüoptionen.

Kurz gesagt liefern die Methoden von *THelloApp* all das, was ein von *TApplication* abgeleitetes Objekt bereitstellen muß: eine Methode, die das Programm initialisiert, eine Methode (den Eventhandler), die auf Ereignisse reagiert, weitere Methoden, die gültige Ereignisse abarbeiten. Diese drei Komponenten sind im Großen und Ganzen alles, was Sie einem Objekt, das von *TApplication* abgeleitet wurde, hinzufügen müssen.

Das Dialogfenster – ein Objekt

Das andere Objekt, auf das wir ebenfalls etwas näher eingehen wollen, ist *TGreetingDialog*; diesem Objekt obliegt die Ausgabe des Dialogfensters.

```
PGreetingDialog = TGreetingDialog;
TGreetingDialog = object(TDialog)
  Button1 : PButton;
  Button2 : PButton;
  Button3 : PButton;
  Button4 : PButton;
  constructor Init;
end;
```

TGreetingDialog ist ein Nachkomme von *TDialog*, das eine allgemeine

Vorlage für Dialogfenster darstellt, in ähnlicher Weise wie *TApplication* eine Vorlage für ein Hauptprogramm bietet. Das Objekt *TDialog* selbst definiert keinerlei interaktive Elemente und ist nichts weiter als ein Rahmen (der sich allerdings durch ziemliche »Intelligenz« auszeichnet), den Sie mit den notwendigen Eingabefeldern, Schaltern und ähnlichen Elementen ausstatten müssen.

TGreetingDialog selbst erweitert *TDialog* um vier Schaltelemente (*Buttons*), die natürlich auch in Turbo Vision definierte Objekte sind (erinnern Sie sich: alle Programmelemente, die irgend etwas auf dem Bildschirm ausgeben wollen, müssen sich dazu eines *View*-Objekts bedienen). Wenn wir von dem Konstruktor absehen, definiert *TGreetingDialog* keine neuen Methoden. All die Eigenschaften, die ein Dialogfenster auszeichnen, darin eingeschlossen der Event-Handler, werden von *TDialog* ererbt.

Programmablauf und Debuggen

Wir haben es schon erwähnt: Turbo-Vision-Programme sind ereignisgesteuert, und solche Programme sind etwas anders strukturiert als konventionelle Programme. Insbesondere trennen ereignisgesteuerte Programme die Programmstrukturen, die Eingaben des Benutzers kontrollieren und bewerten, von den Routinen, die diese Eingaben bearbeiten. Ein konventionelles Programm enthält typischerweise eine große Zahl von Routinen, von denen jede Eingaben entgegennimmt, entscheidet, an welche anderen Teile diese Eingaben weitergereicht werden, schließlich die passenden Routinen aufruft, die ihrerseits diese Eingaben bearbeiten. Diese Routinen geben nach Beendigung ihrer Arbeit die Kontrolle wieder ab, wodurch weitere Eingaben entgegengenommen werden können.

Der zentrale Mechanismus ereignisgesteuerter Programme dagegen besteht darin, Botschaften zu verteilen; der ganze große Rest des Programms braucht sich nicht damit aufzuhalten, Eingaben des Benutzers entgegenzunehmen und zu entscheiden, was damit geschehen soll. Solche Routinen warten vielmehr darauf, daß ihnen durch die zentrale Verteilungsstelle die Eingaben übermittelt werden, für die sie zuständig sind. Das hat Konsequenzen für das Debuggen solcher Programme, denn es ist ziemlich schwierig, den Programmablauf zu verfolgen, um eventuelle Fehler aufzuspüren. Möglicherweise werden Sie dabei anders vorgehen müssen, als Sie es bisher gewohnt waren. In Kapitel 11, »Tips und Tricks«, gehen wir näher auf das Debuggen ein.

Das Hauptprogramm von HELLO

Wie der Hauptprogrammteil eines Turbo-Vision-Programmes aussieht, haben wir weiter vorne schon beschrieben. HELLO macht da keine Ausnahme:

```
var
  Howarya : THelloApp;
begin
  Howarya.Init;
  Howarya.Run;
  Howarya.Done
end;
```

Die erste dieser drei Anweisungen ruft den Konstruktor auf. Jedes Objekt, das virtuelle Methoden enthält, muß auf diese Weise, über den Aufruf des zuständigen Konstruktors initialisiert werden, bevor weitere Methoden dieses Objekts benutzt werden können. Alle in Turbo Vision definierten Konstruktor-Methoden tragen den Namen *Init*. Dieser durchaus sinnvollen Konvention sollten Sie auch in Ihren eigenen Programmen folgen.

Howarya.Init initialisiert das Basis-Objekt des Hauptprogramms, belegt einige wichtige Variablen mit Anfangswerten, löscht den Bildschirm, baut eine gerasterte Arbeitsfläche auf und richtet die Status- und die Menüzeile ein. Weiterhin ruft es die Konstruktor-Methoden einiger weiterer Objekte auf, von denen die meisten nach außen nie in Erscheinung treten: all dies passiert »hinter der Bühne«.

Nutzen Sie den integrierten Debugger, um zu sehen, was dieser Programmteil bewirkt. Drücken Sie [F8], woraufhin *Howarya*.Init ausgeführt wird, und schalten Sie dann über [Alt][F5] auf den Programm-Bildschirm um. Sie sehen eine dreigeteilte Fläche: die (noch) leere Menüzeile, die Arbeitsfläche und die gleichfalls leere Statuszeile. Die Initialisierung des Hauptprogramms über den zuständigen Konstruktor ist eine relativ einfache, geradlinige Agelegenheit. Gleiches gilt für die Destruktor-Methode *Howarya.Done*. Sie ist zuständig für die notwendigen Aufräumungsarbeiten, bevor DOS die Kontrolle wieder übernehmen kann. In einem so einfachen Programm wie HELLO hat der Destruktor nicht allzuviel zu tun, muß aber auch hier dafür sorgen, daß unser Programm das System wieder in den beim Start des Programms vorgefundenen Zustand zurückversetzt.

Die Methode Run

Diese Methode birgt alle Geheimnisse eines Turbo-Vision-Programms. Vergebens werden Sie nach der Definition dieser Methode in HELLO.PAS fahnden – sie ist dort gar nicht definiert. Vielmehr wurde sie vollständig von *TApplication* ererbt.

Gerade in *Run* aber wird Ihr Programm die meiste Zeit verbringen. Im wesentlichen besteht diese Methode aus einer *Repeat-Until*-Schleife. Ihre Darstellung im Pseudo-Code ist unkompliziert:

```
repeat
  Event_entgegennehmen;
  Event_bearbeiten;
until Quit;
```

Dieses abstrakte, von allen Einzelheiten befreite Modell der Methode *Run* zeigt, daß diese Methode fortgesetzt zwei Aufgaben erledigt: ein Ereignis entgegenzunehmen und dieses Ereignis abzuarbeiten. Irgendwann wird eines dieser Ereignisse darauf hinauslaufen, daß die Abbruchbedingung erfüllt und die Schleife verlassen wird: Dadurch ist die Beendigung des Programms eingeleitet.

Die Methode Done

Der Destruktor *Done* ist relativ einfach: Einige Objekte, die zu HELLO gehören – die Statuszeile, die Arbeitsfläche, die Menüzeile – belegen Speicher. *Done* sorgt dafür, daß dieser Speicherplatz wieder freigegeben wird. Außerdem müssen einige Interruptvektoren wieder restauriert werden, worunter sich auch der Vektor befindet, den die Fehlerbehandlungsroutine von Turbo Vision für sich reklamiert hat. Im allgemeinen gilt: die Methode *Done* in Ihrem Programm muß alle speziellen Veränderungen, die Sie über Ihre *Init*-Methode vorgenommen haben, wieder rückgängig machen. Wenn Sie also *TApplication.Init* überschreiben, müssen Sie in der Regel auch die korrespondierende *Done*-Methode überschreiben.

3. Mit Turbo Vision programmieren

Sie haben nun einen ersten Eindruck davon, wie ein Turbo-Vision-Programm aussieht, und wir haben auch in groben Zügen gezeigt, wie ein solches Programm funktioniert. Anhand eines zunächst sehr einfachen Programms, dessen Funktionen wir nach und nach ausbauen und erweitern, zeigen wir in diesem Kapitel, wie Sie Turbo Vision nutzen und die gebotenen Möglichkeiten den Zwecken Ihrer eigenen Programme anpassen können.

Wir werden hier nicht alle Aspekte behandeln können, dazu ist Turbo Vision einfach zu komplex. Lassen Sie sich von dieser Komplexität jedoch nicht einschüchtern. Sicher, hinter Turbo Vision steht ein neues Paradigma der Entwicklung von Programmen, das Ihnen zunächst fremd erscheinen mag. Aber auch hier gilt: Einen Zugang zu neuen Möglichkeiten finden Sie am leichtesten, wenn Sie von diesen Möglichkeiten Gebrauch machen. Der bekannte Aha-Effekt wird dann nicht auf sich warten lassen.

Ihr erstes Programm

Jedes Turbo-Vision-Programm basiert auf einem zentralen Objekt, einem Nachkommen von *TApplication*. Auch unser erstes Programm beginnt damit, ein solches Objekt – wir nennen es *TMyApp* – zu definieren. Die Methoden, die es von *TApplication* ererbt hat, werden wir dann nach und nach überschreiben. Die Variable *MyApp* ist eine Instanz des neuen Objekts *TMyApp*; sie bildet das Fundament unseres Programms.

In diesem Kapitel werden wir uns des öfteren auf das Objekt *MyApp* beziehen; gemeint ist damit Ihre Anwendung, die Instanz eines von *TApplication* abgeleiteten Objekts. Wenn Sie eigene Turbo-Vision-Programme schreiben, werden Sie natürlich andere Namen wählen, Namen, die der Funktion Ihrer Anwendung entsprechen. Wir verwenden *MyApp*, weil diese Benennung kürzer ist als die zugegeben exaktere Redeweise »die Instanz des Objektes, das Sie von *TApplication* abgeleitet haben«.

Mit dem folgenden Code-Abschnitt beginnen wir, ein Beispielprogramm zu entwickeln. Statt nun jedesmal das gesamte Listing abzudrucken, beschränken wir uns darauf, nur jeweils jene Teile zu zeigen, die geändert wurden oder neu hinzugekommen sind. Wenn Sie den Erläuterungen folgen und die

angegebenen Änderungen durchführen, werden Sie bald ein Gefühl dafür bekommen, welch neue Möglichkeiten jeder weitere Schritt eröffnet. Sie sollten sich nicht auf bloßes Nachvollziehen beschränken: Versuchen Sie auch selbst, die Beispielprogramme zu ändern.

Natürlich müssen Sie die Listings nicht selbst eintippen; auf den Originaldisketten finden Sie die besprochenen Beispielprogramme. Die jeweiligen Dateinamen geben wir bei den abgedruckten Programmteilen an.

Der folgende Abschnitt aus TVGUID01.PAS zeigt beispielhaft, wie der zentrale Anweisungsblock eines jeden Turbo-Vision-Programms aussieht:

```
program TFirst;

uses App;

type
  TMyApp = object(TApplication)
  end;

var
  MyApp: TMyApp;

begin
  MyApp.Init;
  MyApp.Run;
  MyApp.Done;
end.
```

Noch haben wir *TMyApp* keine neuen Funktionen hinzugefügt. Normalerweise würden Sie keinen neuen Objekttyp definieren, ohne ihm auch gleich neue Felder und Methoden mitzugeben. Vielmehr könnten Sie die Variable *MyApp* als Instanz von *TApplication* deklarieren. Aber wir wollen unser Programm später erweitern und Voraussetzung dazu ist die Deklaration von *TMyApp* als von *TApplication* abgeleitetes Objekt. Vorerst verhält es sich nicht anders, als eine Instanz von *TApplication* sich auch verhalten würde. Wenn Sie das Programm starten, erscheint der in Abbildung 3.1 gezeigte Bildschirm.

Abbildung 3.1
Der Turbo-Vision-Bildschirm

Und was kann der Benutzer damit anfangen? Vorderhand nur eines: er kann die Tastenkombination [Alt][X] drücken, worauf das Programm endet. Etwas mehr sollte unser Programm schon bieten, und dieses Mehr hängen wir über die Statuszeile oder die Menüleiste in unser Programm ein.

Menüleiste, Arbeitsfläche, Statuszeile

Der Bildschirm, bestehend aus Menüleiste, Arbeitsfläche und Statuszeile, wird von den drei Methoden *InitMenuBar*, *InitDeskTop* und *InitStatusLine* aufgebaut. Keine dieser Methoden müssen Sie selbst aufrufen, *TApplication.Init* erledigt das für Sie. Dagegen muß die *Init*-Methode Ihres Programms als erste Anweisung *TApplication.Init* aufrufen. Beispiel:

```
procedure TMyApp.Init;
begin
  TApplication.Init;
  { Hier folgen die weiteren Initialisierungen,
   die Ihr Programm ben·tigt }
end;
```

Braucht Ihr Programm keine eigenen Initialisierungen, können Sie einfach die aus *TApplication* ererbte *Init*-Methode übernehmen – auch dann, wenn Sie die Methoden *InitMenuBar*, *InitDeskTop* und *InitStatusLine* überschrieben haben. Sowohl sie als auch *Init* selbst sind virtuelle Methoden: *Init* wird immer die richtigen Methoden für die Initialisierung von Statuszeile und Menüleiste aufrufen. Genau dies wird in TVGUID02.PAS vorgeführt.

Die Arbeitsfläche

Obwohl sehr wichtig, brauchen Sie sich um dieses Objekt kaum zu kümmern und es gibt keinen Grund, warum Sie die zuständige Initialisierungsmethode je überschreiben sollten. Die Variable *DeskTop* – eine Instanz des Objekts *TDeskTop* – ist *MyApp* zugeordnet, und jedesmal wenn *MyApp* als Antwort auf die Eingabe des Benutzers ein neues *View*-Objekt instantiiert, ist es die Aufgabe von *DeskTop*, dieses neue Objekt zu verwalten.

Die Statuszeile

zeigt sich zunächst als leerer Balken in der untersten Bildschirmzeile. Das Objekt *TStatusLine* kann allerdings weit mehr als dieser Balken vermuten läßt. Es besitzt die Eigenschaft, Tastenkürzel mit Befehlen verknüpfen und gleichzeitig einen darüber informierenden Text ausgeben zu können. Wenn Sie

diesen Text mit der Maus anklicken oder das Tastenkürzel selbst benutzen, wird der betreffende Befehl abgesetzt.

Das Programm TVGUID02.PAS macht von dieser Möglichkeit Gebrauch. Dazu muß die Methode *TApplication.InitStatusLine* überschrieben werden:

```
procedure TMyApp.InitStatusLine;
var R: TRect;
begin
  GetExtent(R);
  R.A.Y := R.B.Y - 1;
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
      NewStatusKey('~Alt-F3~ Close', kbAltF3, cmClose,
      nil)),
    nil)
  ));
end;
```

Natürlich müssen Sie diese neue Methode in den Deklarationsteil von *TMyApp* einfügen. Auch die *Uses*-Anweisung ist zu erweitern. Da wir eine neue Statuszeile aufbauen, dabei auf Standard-Tastendefinitionen zugreifen und demnächst auch Menüs definieren, müssen wir die Units *Objects*, *Menus* und *Drivers* zusätzlich zu *App* einbinden.

Die Initialisierung besteht in einer verschachtelten Sequenz, innerhalb der die Funktionen NewStatusDef und *NewStatusKey* aufgerufen werden. Näher beschrieben werden beide Funktionen in Kapitel 15 »Außerhalb der Objekte«. Der in Turbo Vision definierte Standardbefehl *cmQuit* wird mit der Taste [Alt][X] und der Befehl *cmClose* mit der Tastenkombination [Alt][F3] verbunden. Ein Hinweis zur Namensgebung: Die Benennungen aller in Turbo Vision definierten Befehle beginnen mit der Buchstabensequenz *cm*.

Der letzte Eintrag in der Statuszeile lautet »Alt-F3 Close«. Der zwischen den Tilde-Zeichen stehende Teil wird hervorgehoben ausgegeben, und wenn der Benutzer diesen String mit der Maus anklickt, wird der angegebene Befehl ausgeführt.

Wenn Sie jedoch das Programm TVGUID02 ausführen, werden Sie feststellen, daß diese Option keineswegs hervorgehoben erscheint; auch ein Mausklick zeigt keine Wirkung. Denn dieser Befehl ist zunächst nicht aktiviert, und das zeigt sich auch in der Form seiner Darstellung auf dem Bildschirm. Das ist sinnvoll: wo noch kein Fenster geöffnet wurde, kann auch keines geschlossen werden. Sobald Sie ein Fenster öffnen, wird *cmClose* aktiviert und der Hinweis auf das Tastenkürzel hervorgehoben dargestellt.

Sie sehen: Haben Sie das Objekt *StatusLine* einmal initialisiert, dann ist Ihre Arbeit getan. Das Objekt reagiert selbständig auf Eingaben des Benutzers, ohne daß Sie sich darum kümmern müßten.

Neue Befehle definieren

Die Befehle *cmClose* und *cmQuit*, die wir oben verwendet haben, mußten wir nicht selbst definieren, es handelt sich um Standardbefehle, die Turbo

Vision bereitstellt. Die Befehle, die Sie neu einführen, müssen Sie als Konstanten definieren. Beachten Sie aber, daß Turbo Vision einige Konstantenwerte für eigene Zwecke reserviert. Vergleichen Sie dazu den Abschnitt »Befehle definieren« in Kapitel 6, »Ereignisgesteuerte Programme«. Zeigen wir das am Beispiel eines Befehls, über den ein Fenster geöffnet wird:

```
const
  cmNewWin = 199;
```

Diesen Befehl verknüpfen wir mit dem Tastenkürzel [F4] und lassen die neue Option in der Statuszeile anzeigen:

```
StatusLine := New(PStatusLine, Init(R,
  NewStatusDef(0, $FFFF,
    NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
    NewStatusKey('~F4~ New', kbF4, cmNewWin,
      NewStatusKey('~Alt-F3~ Close', kbAltF3, cmClose,
    nil))),
  nil)
));
```

Diese Anweisung zur Initialisierung der Statuszeile stellt eine gute Einführung in die Syntax zur Initialisierung der Menüleiste dar. Allerdings ist diese noch ein wenig komplexer.

Die Menüleiste

wird initialisiert über ineinander verschachtelte Aufrufe der in Turbo Vision definierten Funktionen *NewMenu*, *NewSubMenu*, *NewItem* und *NewLine*.

Was wir schon bei der Statuszeile schätzen lernten, gilt auch für die Menüleiste: Mit der Initialisierung ist Ihre Arbeit beendet. Denn auch die Menüleiste versteht, auf Eingaben des Benutzers zu reagieren. Das folgende Beispiel definiert ein einfaches Menü, dessen einziger Eintrag ein Untermenü ausgibt, das ebenfalls nur eine Option anbietet:

```
const
  cmFileOpen = 200;

procedure TMyApp.InitMenuBar;
var R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~F~ile', hcNoContext, NewMenu(
      NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,
      nil)),
    nil)
```

```
  )));
end;
```

Eine Bemerkung zu dem Aufrufparameter von *NewSubMenu* und *NewItem*, der im oben gezeigten Beispiel mit dem Wert *hcNoContext* belegt wurde: Jeder Menüoption kann ein numerischer Wert zugeordnet werden, was die Funktion eines kontextsensitiven Hilfesystems, das Sie vielleicht in Ihre Programme einbauen wollen, beträchtlich erleichtert. Denn über diesen Wert kann Ihr Hilfesystem sehr leicht ermitteln, aus welchem Programmkontext heraus der Benutzer Hilfe angefordert hat. Im Beispiel oben wird einfach die mit dem Wert 0 belegte Konstante *hcNoContext* übergeben. Wollen Sie ein Hilfesystem installieren, müssen Sie hier natürlich Werte übergeben, die Ihrem Programm erlauben, die zu dem jeweiligen Menüpunkt passende Hilfestellung auszugeben.

Um nun das Menü *File* um einen zweiten Auswahlpunkt zu erweitern, fügen wir einfach einen weiteren Aufruf der Funktion *NewItem* ein:

```
MenuBar := New(PMenuBar, Init(R, NewMenu(
  NewSubMenu('~F~ile', hcNoContext, NewMenu(
    NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,
    NewItem('~N~ew', 'F4', kbF4, cmNewWin, hcNoContext,
    nil))),
  nil)
)));
```

Einen weiteren Eintrag in der Menüleiste erhalten wir über einen erneuten Aufruf der Funktion *NewSubMenu*:

```
MenuBar := New(PMenuBar, Init(R, NewMenu(
  NewSubMenu('~F~ile', hcNoContext, NewMenu(
    NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,
    NewItem('~N~ew', 'F4', kbF4, cmNewWin, hcNoContext,
    nil))),  { schlieøende Klammern f¡r Men¡wahl }
  NewSubMenu('~W~indow', hcNoContext, NewMenu(
    NewItem('~N~ext', 'F6', kbF6, cmNext, hcNoContext,
    NewItem('~Z~oom', 'F5', kbF5, cmZoom, hcNoContext,
    nil)),
  nil)))  { schlieøende Klammern f¡r Men¡s }
)));
```

Hier haben wir zwei weitere in Turbo Vision definierte Befehle – *cmNext* und *cmZoom* – mit Menüoptionen und auch mit Tastenkürzeln verknüpft.

Wir können Menüoptionen durch eine horizontale Linie voneinander trennen. Dazu müssen wir zwischen den Aufrufen der Funktion *NewItem* die Funktion *NewLine* aufrufen:

```
MenuBar := New(PMenuBar, Init(R, NewMenu(
  NewSubMenu('~F~ile', hcNoContext, NewMenu(
    NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,
    NewItem('~N~ew', 'F4', kbF4, cmNewWin, hcNoContext,
    NewLine(
```

```
      NewItem('E~x~it', 'Alt-X', kbAltX, cmQuit, hcNoContext,
      nil))))),
    NewSubMenu('~W~indow', hcNoContext, NewMenu(
      NewItem('~N~ext', 'F6', kbF6, cmNext, hcNoContext,
      NewItem('~Z~oom', 'F5', kbF5, cmZoom, hcNoContext,
      nil))),
    nil))
));
```

Wir haben nun eine Anzahl von Befehlen definiert, aber die meisten sind leider nicht aktiviert; schlimmer noch: *cmNewWin* und *cmFileOpen* führen überhaupt noch keine Aktion aus.

Kein Grund zur Enttäuschung: Wir haben schon eine ganze Menge erreicht. Einer der großen Vorteile, die ereignisgesteuerte Programme bieten, besteht in der Trennung der Programmteile, die Eingaben des Benutzers entgegennehmen, von den Teilen des Programms, die auf diese Eingaben reagieren oder sie verarbeiten.

Hätten wir überkommene Programmiertechniken genutzt, müßten wir den Programmcode, den wir gerade geschrieben haben, erneut überarbeiten, da und dort Anweisungen einfügen, die zum Beispiel ein Fenster öffnen oder auch andere Aktionen durchführen. Das können wir uns jetzt sparen. Denn wir haben einen Mechanismus aufgebaut, der in der Lage ist, Eingaben entgegenzunehmen und passende Befehle abzusetzen. Das Einzige, was uns noch zu tun bleibt: Wir müssen ein paar Routinen schreiben, die auf diese Befehle reagieren. Darauf werden wir im nächsten Abschnitt eingehen.

Wir öffnen ein Fenster

In Turbo Vision ist natürlich auch ein Bildschirmfenster ein Objekt, und zwar eines mit bemerkenswerten Eigenschaften. Ein solches Fenster kann sich öffnen, seine Größe und seine Position verändern, sich wieder schließen, all dies in Reaktion auf Eingaben des Benutzers. Dazu müssen Sie keine einzige Programmzeile schreiben. Vielleicht sehen Sie in einem Bildschirmfenster einen Rahmen, innerhalb dessen Sie Informationen darstellen können. Das ist nicht ganz so. Besser Sie stellen sich unter einem Fenster eine Art Bühne vor, eine Bühne, auf der andere Objekte sich darstellen. Diese Objekte, die wir in diesem Buch als *Views* bezeichnen, sind dem Fensterobjekt zugeordnet, sie gehören zu dem Objekt Fenster.

Daraus ergeben sich zwei mögliche Wege, ein Fenster zu definieren. Sie legen zuerst fest, was überhaupt auf dem Bildschirm ausgegeben werden soll, definieren die entsprechenden Objekte, die *Views*, und initialisieren dann ein Fenster, innerhalb dessen diese Objekte dann erscheinen. Das ist der Weg, den Sie wahrscheinlich beschreiten, wenn Sie eigene Programme entwickeln.

In dem folgenden Programm – Sie finden den Quellcode in der Datei TVGUID04.PAS – gehen wir anders vor. Wir definieren zuerst ein Fenster-

objekt, das wir von *TWindow* ableiten, und verknüpfen dieses Objekt mit der Arbeitsfläche. Vorderhand verfügt unser Fensterobjekt nur über die von *TWindow* ererbten Methoden, später werden wir einige neue hinzufügen.

```
uses Views;

const
  WinCount: Integer = 0;

type
  PDemoWindow = ^TDemoWindow;
  TDemoWindow = object(TWindow)
  end;

procedure TMyApp.NewWindow;
var
  Window: PDemoWindow;
  R: TRect;
begin
  Inc(WinCount);
  R.Assign(0, 0, 21, 7);
  R.Move(Random(58), Random(16));
  Window := New(PDemoWindow, Init(R, 'Demo Window', WinCount));
  DeskTop^.Insert(Window);
end;

procedure TMyApp.HandleEvent(var Event: TEvent);
begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmNewWin: NewWindow;
    else
      Exit;
    end;
    ClearEvent(Event);
  end;
end;
```

Wir können von dem Fensterobjekt erst dann Gebrauch machen, wenn wir den Befehl *cmNewWin* mit einer Menüoption oder einem Tastenkürzel verknüpft haben. Aber das ist ja schon geschehen. Wenn der Benutzer *cmNewWin* aufruft, übergibt Turbo Vision diesen Befehl an *TMyApp.HandleEvent*, das seinerseits *TMyApp.NewWindow* zur Ausführung des Befehls aufruft. Betrachten wir diese beiden neuen Methoden näher.

Ein Fenster definieren

Bei der Initialisierung eines Bildschirmfensters übergeben Sie drei Parameter: Größe und Position des Fensters, Fenstertitel und Fensternummer.

Der erste Parameter, der die Position und die Größe eines Fensters bestimmt, ist ein Objekt des Typs *TRect*, das einen rechteckigen Bereich beschreibt. Dessen Methode *Assign* ordnet diesem Bereich den Ausgangspunkt und die Größe zu. Die notwendigen Parameter beschreiben dessen obere linke und die untere rechte Ecke. In unserem Beispiel beginnt dieses Rechteck mit den Ursprungskoordinaten. Das verwendete Koordinatensystem sowie das Objekt *TRect* behandeln wir in Kapitel 5, »View-Objekte«, ausführlicher.

Über einen Aufruf der Methode *Move* ändern wir mittels zweier Zufallswerte die Position des Rechtecks. Das Objekt *TRect* kennt verschiedene Methoden, mithilfe derer die Werte für Position und Größe geändert werden. Vergleichen Sie dazu den Referenzteil dieses Bandes.

Der zweite Initialisierungsparameter ist ein String, der in den Rahmen des Fensters eingeblendet wird. Über ihn können Sie Ihr Fenster mit einem Titel versehen.

Der dritte Parameter wird im Feld *Number* abgelegt. Hat dieser Parameter einen Wert zwischen 1 und 9, dann erscheint diese Zahl im oberen rechten Teil des Rahmens. In diesem Fall können Sie das betreffende Fenster über die Tastenkombinationen `[Alt][1]` bis `[Alt][9]` anwählen. Wollen Sie dem Fenster keine Nummer zuordnen, dann übergeben Sie stattdessen einfach die in Turbo Vision definierte Konstante *wnNoNumber*.

Nach der Initialisierung verknüpfen wir das Fenster mit dem Objekt *DeskTop*. Wenn Sie die Anweisung

```
DeskTop^.Insert(Window);
```

ausführen lassen, dann wird einerseits das Fenster dargestellt und andererseits mit der Arbeitsfläche verbunden. Es wird zu einem Element der Arbeitsfläche. Über die Methode *Insert* eignen sich *View*-Objekte sozusagen andere Objekte an; in unserem Fall ergreift das Objekt *DeskTop* Besitz von dem Objekt *Window*. Dies wiederum kann seinerseits, wie wir noch sehen werden, weitere Darstellungselemente besitzen. Auf diese Weise konstituiert sich eine baumartig strukturierte Ordnung der verschiedenen Darstellungsobjekte. Sie alle sind über unterschiedlich komplexe Verzweigungen mit dem zentralen Objekt eines jeden Turbo-Vision-Programms verbunden: dem Objekt, das von *TApplication* abgeleitet wurde.

Nur Objekte, die direkt oder indirekt von dem Objekt *TGroup* abstammen, verfügen über diese *Insert*-Methode. Daher sind auch nur sie in der Lage, mit anderen *View*-Objekten zusammen eine Gruppe zu bilden. Das gilt beispielsweise für die Objekte *TWindow*, *TDeskTop* und natürlich für *TApplication*. Wenn wir im folgenden von Gruppe sprechen, meinen wir damit ein Objekt, das über andere Objekte verfügt, wodurch auch komplexe Darstellungen auf dem Bildschirm möglich werden.

Zurück zu dem Fenster, das wir gerade geöffnet haben. Um es wieder zu schließen, müssen wir unserem Programm keine weiteren Anweisungen hin-

zufügen. Wenn der Benutzer das Schließfeld des Fensters anklickt, übernimmt Turbo Vision den Rest. Diese Eingabe mittels der Maus setzt übrigens ebenfalls den Befehl *cmClose* ab, den Befehl, den wir auch über die Statuszeile und das Tastenkürzel [Alt][F3] dem Benutzer zur Verfügung stellten. Wie schon erwähnt, ist dieser Befehl zunächst nicht aktiviert. Wird aber ein Fenster geöffnet – entweder über [F3] oder die Menüoptionen File / Open – aktiviert sich automatisch auch dieser Befehl.

Eigenschaften eines Fensters

Nehmen Sie sich die Zeit und experimentieren ein wenig mit dem Programm, das wir gerade entwickeln. Es verfügt bereits über einige beeindruckende Funktionen. Es kann einander überlappende Fenster öffnen, deren Größe oder Position verändern, sie wieder schließen. Für weniger als 100 Zeilen Programmcode gar nicht schlecht.

Wenn Sie den Befehl *cmNewWin* mehrmals absetzen, erzeugen Sie mehrere Fenster über der Arbeitsfläche, jedes mit einer individuellen Nummer. Jedesmal wenn *TMyApp* ein neues Fenster initialisiert, ordnet es dieses Fenster der Arbeitsfläche zu, deren Aufgabe es ist, diese Fenster zu verwalten und Ereignisse, die ein Fenster betreffen, an dieses weiterzuleiten. Wenn Sie also zum Beispiel mittels der Maus irgend etwas eingeben, wird diese Eingabe als Ereignis aufgefaßt und an das aktive Fenster weitergegeben.

Übrigens könnten Sie die Positions- und Größenparameter für ein Fenster mit so unsinnigen Werten wie 190, 190, 210, 197 belegen. Auch sie würden akzeptiert, und das so initialisierte Fenster der Arbeitsfläche zugeordnet. Allerdings wäre ein solches Fenster nie zu sehen, und es gibt kein Mittel, es in den sichtbaren Bereich der Arbeitsfläche zu bewegen.

Ein Objekt des Typs *TWindow* ist eine Gruppe, die aber zum Zeitpunkt ihrer Initialisierung nur über ein einziges weiteres *View*-Objekt, über das Objekt *TFrame* verfügt, dessen Aufgabe darin besteht, das Fenster mit einem Rahmen zu umgeben. Diesen Rahmen klickt der Benutzer an, wenn er das Fenster

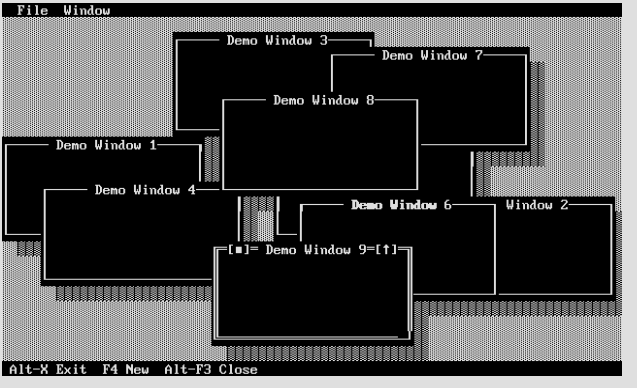


Abbildung 3.2 Mehrere Fenster geöffnet

bewegen, seine Größe verändern oder es schließen will. Auch die Ausgabe des Fenstertitels, den Sie bei der Initialisierung des Fensters angegeben haben, gehört zu den Aufgaben dieses Objekts.

Innenleben eines Fensters

Hätten wir es mit traditionellen Bildschirmfenstern zu tun, würden wir als nächstes irgend etwas innerhalb dieses Fensters ausgeben. Ein *TWindow* ist jedoch kein leeres Blatt, das darauf wartet, beschrieben zu werden. Es ist eine Gruppe, von der zunächst nicht weiter zu sehen ist als der zugehörige Rahmen. Um innerhalb dieses Rahmens etwas darstellen zu können, müssen wir andere Objekte bemühen.

Das folgende Beispielprogramm (das Sie in TVGUID05.PAS finden), definiert ein neues Objekt *TInterior*, das von *TView* abgeleitet wird. Dieses Objekt überdeckt zunächst den gesamten innerhalb des Fensters zur Verfügung stehenden Raum. Später werden wir es in seiner Größe etwas einschränken, um Platz zu schaffen für die Darstellung anderer Objekte. Einem Fenster können Sie so viele *View*-Objekte zuordnen, wie Sie für notwendig halten: Eingabefelder, Schalter, Auswahlboxen. Und Sie werden sehen, wie einfach Sie den Rahmen des Fensters durch Rollbalken überlagern können.

Ihre *View*-Objekte können Sie wohlgeordnet ausgerichtet oder einander überlappend darstellen lassen. Wie sie zusammenarbeiten und sich gegenseitig respektieren, ist Ihnen überlassen.

Das Objekt *TInterior*, das wir kreieren, ist ein einfacher Nachkomme von *TView*, erbt von diesem aber auch die Fähigkeit, über einen Rahmen verfügen zu können – wenn Sie das wollen. Verglichen mit dem Rahmen eines *TWindow*-Objekts hat dieser Rahmen eher ornamentalen Charakter: Wenn Sie ihn anklicken, wird keine Aktion ausgelöst. Aber auch er ist davor geschützt, durch Text, der innerhalb des Rahmens ausgegeben wird, überschrieben zu werden.

Solange das *TView*-Objekt, wie in unserem Falle, das gesamte Innere eines Fensters ausfüllt, ist der Rahmen auch nicht sichtbar. Er wird vom Rahmen des Fenster überdeckt. Sichtbar wird er erst, wenn das Objekt kleiner als der zur Verfügung stehende Raum ist. Damit bietet sich die Möglichkeit, mit Hilfe eines solchen Rahmens verschiedene *View*-Objekte innerhalb des gleichen Fensters optisch voneinander abzugrenzen.

Im folgenden Ausschnitt aus TVGUID05.PAS statten wir *TInterior* mit einem Rahmen aus:

```
PInterior = ^TInterior;
  TInterior = object(TView)
    constructor Init(var Bounds: TRect);
    procedure Draw; virtual;
  end;
```

```
constructor TInterior.Init(var Bounds: TRect);
begin
  TView.Init(Bounds);
  GrowMode := gfGrowHiX + gfGrowHiY;
  Options := Options or ofFramed;  { ergibt einen Rahmen }
end;

procedure TInterior.Draw;
const
  Greeting: string = 'Hello, World!';
begin
  TView.Draw;
  WriteStr(4, 2, Greeting,$01);
end;

constructor TDemoWindow.Init(Bounds: TRect; WinTitle: String;
  WindowNo: Word);
var
  S: string[3];
  Interior: PInterior;
begin
  Str(WindowNo, S);
  TWindow.Init(Bounds, WinTitle + ' ' + S, wnNoNumber);
  GetClipRect(Bounds);
  Bounds.Grow(-1,-1);
  Interior := New(PInterior, Init(Bounds));
  Insert(Interior);
end;
```

Abbildung 3.3
Das Fenster in TVGUID05

Was zu sehen ist

Alle *View*-Objekte in Turbo Vision haben die Fähigkeit, sich selber auf dem Bildschirm darstellen zu können. Diese Fähigkeit erwerben sie über die Methode *Draw*. Wenn Sie ein Objekt als Nachkomme eines dieser *View*-Objekte definieren, müssen Sie dessen *Draw*-Methode überschreiben, damit auch Ihr Objekt die Fähigkeit zur Selbstdarstellung erhält. *TInterior* ist ein Nachkomme von *TView*, weshalb wir eine neue *Draw*-Methode bereitstellen müssen.

Die neue Methode *TInterior.Draw* ruft zunächst die *Draw*-Methode seines Vorfahren *TView* auf, die sich darauf beschränkt, den Bereich, den unser Objekt einnimmt, zu löschen. Dieses Verfahren ist zur Nachahmung nicht empfohlen; die neue *Draw*-Methode sollte die Verantwortung für den gesamten Bereich des Objekts übernehmen. Dann wird es natürlich überflüssig, *TView.Draw* aufzurufen. Wenn Sie sich dennoch auf diese Methode stützen wollen, müssen Sie in Kauf nehmen, daß der Bildschirm flackert, da dann Teile von *TInterior* mehr als einmal ausgegeben werden müssen.

Wenn Sie sehen wollen, warum eine *Draw*-Methode den übernommenen Bereich in eigener Verantwortung verwalten sollte, dann kommentieren Sie den Aufruf von *TView.Draw* aus, kompilieren das Programm und starten es. Bewegen Sie dann das Fenster über den Bildschirm und ändern Sie auch seine Größe.

Turbo Vision ruft die *Draw*-Methode eines *View*-Objekts immer dann auf, wenn dieses Objekt auf dem Bildschirm ausgegeben, in seiner Größe verändert oder über den Bildschirm bewegt wird. Wenn Sie wollen, daß ein Objekt sich auf dem Bildschirm darstellt, rufen Sie die Methode *DrawView* statt *Draw* auf. *DrawView* sorgt dafür, daß nur jene Teile des Objekts auf dem Bildschirm erscheinen, die dort Platz finden. Das ist sehr wichtig: Sie überschreiben die Methode *Draw*, rufen sie aber niemals selbst auf; Sie rufen *DrawView* auf, dürfen aber diese Methode niemals überschreiben.

Innerhalb der neuen Methode *TInterior.Draw* rufen wir die Prozedur *WriteStr* auf. Warum verwenden wir nicht einfach die Standard-Prozedur *Write*? Einfach deshalb, weil *Write* und *WriteLn* in einem Turbo-Vision-Programm nichts zu suchen haben. Denn erstens würden Informationen, die Sie über diese Prozeduren auf den Bildschirm schreiben, wahrscheinlich sehr bald zerstört werden. Zweitens können beide Prozeduren die innerhalb eines Objekts lokal gültigen Koordinaten für die Ausgabeposition nicht berücksichtigen und haben drittens keine Möglichkeit, die Farbe, in der ausgedruckt wird, zu bestimmen. Die Prozedur *WriteStr* kennt dagegen diese Nachteile nicht und kann darüber hinaus auch die Ausdehnung des jeweiligen Objekts berücksichtigen. Als Parameter benötigt *WriteStr* die Angabe der X- und der Y-Koordinate, den auszugebenden String und einen numerischen Wert, der als Index in der gerade aktuellen Farbpalette verwendet wird.

Textdateien betrachten

Die Datei TVGUID06.PAS enthält den Quellcode eines Programms, das eine Textdatei einliest, und das Objekt *TInterior* dazu nutzt, den Inhalt dieser Datei darzustellen. Lassen Sie sich nicht davon irritieren, daß dabei auch einiger »Müll« auf dem Bildschirm erscheint. Auf diesen Müll und seine Ursache kommen wir noch zu sprechen.

```
const
  MaxLines = 100; { willk¡rlicher Wert }

var
  LineCount: Integer;
  Lines: array[0..MaxLines - 1] of PString;

type
  PInterior = ^TInterior;
  TInterior = object(TView)
    constructor Init(var Bounds: TRect);
    procedure Draw; virtual;
  end;

procedure TInterior.Draw;
var
  Y: Integer;
begin
  TView.Draw;
  for Y := 0 to Size.Y - 1 do
  begin
    WriteStr(0, Y, Lines[Y]^, $01);
  end;
end;

procedure ReadFile;
var
  F: Text;
  S: String;
begin
  LineCount := 0;
  Assign(F, ParamStr(1));
  Reset(F);
  while not Eof(F) and (LineCount < MaxLines) do
  begin
    ReadLn(F, S);
    Lines[LineCount] := NewStr(S);
    Inc(LineCount);
  end;
  Close(F);
end;
```

```
procedure DoneFile;
var
  I: Integer;
begin
  for I := 0 to LineCount do
    if Lines[I] <> nil then DisposeStr(Lines[I]);
end;
```

Die Prozedur *ReadFile* öffnet eine Textdatei und überträgt den darin gespeicherten Text in das Array *Lines*. Der in Turbo Vision global verfügbare Typ *PString*, auf dem das Array basiert, ist als Zeiger auf einen String definiert. Turbo Vision stellt auch die Funktion *NewStr* bereit, die einen String auf dem Heap speichert und einen Zeiger auf diesen String zurückgibt. Der Widerpart dieser Funktion bildet die Prozedur *DisposeStr*, deren Aufgabe darin besteht, den von einem String belegten Heapspeicher wieder freizugeben. Verwenden Sie zu diesem Zweck auf keinen Fall die Standardprozedur *Dispose*!

Kommen wir nun auf den schon erwähnten Müll zu sprechen. Überall dort, wo eine leere Textzeile dargestellt werden sollte, erscheint stattdessen dieser Müll. Schuld daran ist unsere *Draw*-Methode. Sie entspricht nicht der vorhin erhobenen Forderung, wonach eine solche Methode den gesamten Bereich verwalten sollte, für den das jeweilige *View*-Objekt verantwortlich ist.

Die bisherige *Draw*-Methode beschränkt sich darauf, über *WriteStr* die einzelnen Zeilen des Textes auszugeben. Diese Zeilen können aber oft den zur Verfügung stehenden Raum nicht überdecken, zum Beispiel dann, wenn eine Zeile überhaupt kein Zeichen enthält. Solche Zeilen müßten also mit Leerzeichen erst auf die erforderliche Breite aufgebläht werden.

In den Puffer umleiten

Die neue *Draw*-Methode verwendet einen Puffer des Typs *TDrawBuffer*, der zunächst mit den gewünschten Werten – Leerzeichen plus Attributbyte – belegt, und in den dann die Zeichenkette hineinkopiert wird. Der Typ *TDrawBuffer* ist in Turbo Vision global verfügbar und folgendermaßen definiert:

```
TDrawBuffer = array[0..MaxViewWidth-1] of Word;
```

Die Konstante *MaxViewWidth* hat den Wert 132, wodurch gewährleistet ist, daß der Puffer ausreichend groß ist, um in jedem möglichen Textmodus eine Bildschirmzeile überdecken zu können. Jedes *Word* dieses Puffers kann ein Zeichen plus das zugehörige Attribut-Byte aufnehmen.

Und so sieht die neue Methode *Draw* aus:

```
procedure TInterior.Draw;
var
  Color : Byte
  Y: Integer;
  B : TDrawBuffer;
begin
```

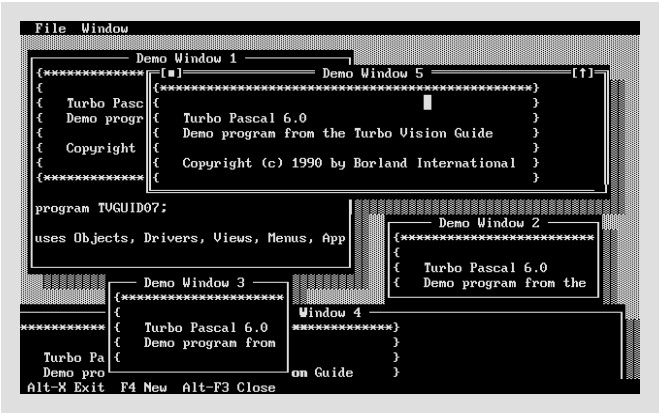


Abbildung 3.4 Mehrere Textfenster geöffnet

```
    Color := GetColor(1);
    for Y := 0 to Size.Y - 1 do
    begin
      MoveChar(B, ' ', Color, Size.X);
      if (Y < LineCount) and (Lines[Y] <> nil then
        MoveStr(B, Copy(Lines[Y]^, 1, Size.X), Color);
      WriteLine(0, Y, Size.X, 1, B);
    end;
  end;
```

Zunächst wird der Puffer mit der erforderlichen Zahl von Leerzeichen plus dem aktuellen Farbwert aufgefüllt; der Parameter *Size.X* enthält die horizontale Ausdehnung des *View*-Objekts. Darauf rufen wir *MoveStr* auf, um die Zeichenkette in den Buffer zu übertragen; diesen Puffer an der richtigen Stelle in den Bildschirmspeicher zu kopieren, ist schließlich die Aufgabe der Prozedur *WriteLine*.

Den Puffer füllen

Turbo Vision stellt drei Prozeduren zur Verfügung, an deren Namen zu erkennen ist, welcher Datentyp jeweils mit ihrer Hilfe in einen *TDrawBuffer* kopiert werden kann:

```
procedure MoveChar(var Dest; C: Char; Attr: Byte; Count : Word);
procedure MoveStr(var Dest; Str: String; Attr: Byte);
procedure MoveBuf(var Dest; var Source; Attr: Byte; Count : Word);
```

Auf dem Bildschirm ausgeben

Über die schon erwähnten Prozeduren *WriteStr* und *WriteLine* hinaus, kennt Turbo Vision zur Ausgabe auf dem Bildschirm noch *WriteChar*, womit genau ein Zeichen auch wiederholt geschrieben wird, und *WriteBuf*, das den Inhalt des Puffers ausgibt. Alle vier Prozeduren sind Methoden des Objekts *TView*.

```
procedure WriteChar(X,Y: Integer; C: Char; Color: Byte; Count:
                    Integer);
procedure WriteStr(X,Y: Integer; Str: String; Color: Byte);
procedure WriteLine(X,Y,W,H: Integer; var Buf);
procedure WriteBuf(X,Y,W,H: Integer; var Buf);
```

In *TInterior.Draw* verwenden wir *WriteLine* dazu, eine Zeile genau einmal auszugeben. Über den Parameter *H* können Sie angeben, wie oft die Ausgabe der Zeichenkette wiederholt werden soll. Wenn wir also voraussetzen, daß der übergebene Puffer die Zeichenkette »Hello, World!« enthält, wird ein Aufruf von *WriteLine*(*0,0,13,4*) die folgende Ausgabe erzeugen:

```
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

Die Prozedur *WriteBuf* ist in der Lage, den Inhalt des Puffers in einer bestimmten Breite zu umbrechen. Die Breite legen Sie über den Parameter *W* fest. Angenommen, der Puffer enthielte die Zeichen »ABCDEFGHIJKLMNOP«, und Sie rufen *WriteBuf*(*0,0,4,4*) auf, dann erhalten Sie folgende Ausgabe:

```
ABCD
EFGH
IJKL
MNOP
```

Diese vier Ausgabeprozeduren sollten nur innerhalb einer *Draw*-Methode aufgerufen werden. Das klingt wie eine Einschränkung, ist aber keine: In Turbo Vision ist es diesen *Draw*-Methoden vorbehalten, etwas auf dem Bildschirm darstellen.

Wieviel ist zu schreiben?

Unsere Methode *TInterior.Draw* schreibt gerade soviel von dem Text, wie in den Innenraum des Fensters paßt. Andernfalls verbrächte *Draw* die meiste Zeit mit dem Versuch, Text auszugeben, der aber an den Grenzen von *TInterior* abgeschnitten würde.

Benötigt ein *View*-Objekt sehr viel Zeit, sich selbst darzustellen, können Sie zunächst *GetClipRect* aufrufen. Sie erhalten dann die Größe des Rechtecks zurück, das unbedingt neu aufgebaut werden muß, also nur der Teil des von dem Objekt verwalteten Bereichs, der tatsächlich auf dem Bildschirm sichtbar ist. Sagen wir, Sie hätten ein sehr komplexes Dialogfenster, das Sie aber zum größten Teil aus dem Bildschirm hinausgeschoben haben, um die Objekte unter diesem Fenster zu sehen. Wenn Sie nun *GetClipRect* aufrufen, bevor das Dialogfenster neu ausgegeben wird, ersparen Sie *Draw*, auch diejenigen Teile des Fensters auszugeben, die gar nicht mehr auf den Bildschirm passen.

Das Fenster über den Text bewegen

Ein Programm, das für sich in Anspruch nimmt, Textdateien auf dem Bildschirm ausgeben zu können, und das dann nur deren erste Zeilen darstellt, ist ziemlich nutzlos. Wir wollen auch andere Teile des Textes sehen, und dazu ist es notwendig, das Fenster mit der Fähigkeit auszustatten, sich über diesen Text bewegen zu können. In TVGUID08.PAS wird das Objekt *TInterior* um diese Fähigkeit bereichert. Dazu ist es notwendig, in die Abstammungsverhältnisse von *TInterior* einzugreifen: Wir leiten dieses Objekt jetzt von *TScroller* ab, denn wir wollen unser *View*-Objekt mit Rollbalken ausstatten. Weiter erweitern wir *TDemoWindow* um die Methode *MakeInterior,* und erreichen damit, daß eine Methode das Öffnen des Fensters und eine andere die Darstellung der Rollbalken besorgt.

```
type
  PInterior = ^TInterior;
  TInterior = object(TScroller)
    constructor Init(var Bounds: TRect; AHScrollBar,
      AVScrollBar: PScrollBar);
    procedure Draw; virtual;
  end;

  PDemoWindow = ^TDemoWindow;
  TDemoWindow = object(TWindow)
    constructor Init(Bounds: TRect; WinTitle: String; WindowNo:
                     Word);
    procedure MakeInterior(Bounds: TRect);
  end;

constructor TInterior.Init(var Bounds: TRect; AHScrollBar,
  AVScrollBar: PScrollBar);
begin
  TScroller.Init(Bounds, AHScrollBar, AVScrollBar);
  GrowMode := gfGrowHiX + gfGrowHiY;
  Options := Options or ofFramed;
  SetLimit(128, LineCount);
end;

procedure TInterior.Draw;
var
  Color: Byte;
  I, Y: Integer;
  B: TDrawBuffer;
begin
  Color := GetColor(1);
  for Y := 0 to Size.Y - 1 do
  begin
    MoveChar(B, ' ', Color, Size.X);
    i := Delta.Y + Y;
```

```
    if (I < LineCount) and (Lines[I] <> nil) then
      MoveStr(B, Copy(Lines[I]^, Delta.X + 1, Size.X), Color);
    WriteLine(0, Y, Size.X, 1, B);
  end;
end;

procedure TDemoWindow.MakeInterior(Bounds: TRect);
var
  HScrollBar, VScrollBar: PScrollBar;
  Interior: PInterior;
  R: TRect;
begin
  VScrollBar := StandardScrollBar(sbVertical + sbHandleKeyboard);
  HScrollBar := StandardScrollBar(sbHorizontal + sbHandleKeyboard);
  GetExtent(Bounds);
  Bounds.Grow(-1,-1);
  Interior := New(PInterior, Init(Bounds, HScrollBar, VScrollBar));
  Insert(Interior);
end;

constructor TDemoWindow.Init(Bounds: TRect; WinTitle: String;
  WindowNo: Word);
var
  S: string[3];
begin
  Str(WindowNo, S);
  TWindow.Init(Bounds, WinTitle + ' ' + S, wnNoNumber);
  MakeInterior(Bounds);
end;
```

Die Methode *TDemoWindow.MakeInterior* initialisiert den vertikalen und den horizontalen Rollbalken und ordnet sie dem Fensterobjekt zu.

Ein von *TScroller* abgeleitetes *View*-Objekt verhält sich sozusagen wie ein Guckloch, das Sie über einen wesentlich größeres Datenbereich verschieben

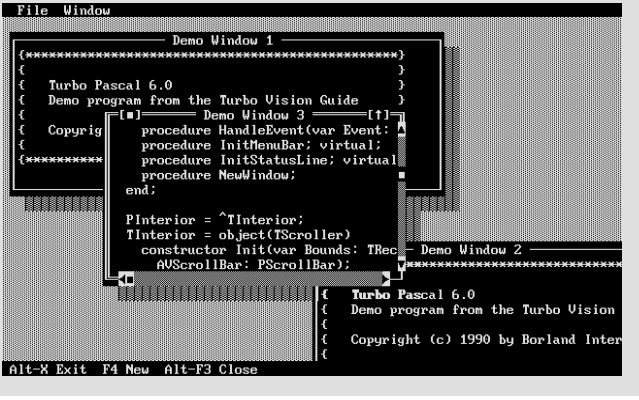


Abbildung 3.5 Gucklöcher auf Textdateien

können, um immer andere Teile dieses Bereichs zu sehen. Die Markierungen auf den Rollbalken symbolisieren die relative Position unseres Gucklochs innerhalb des Datenbereichs. Die Variablen *Delta.X* und *Delta.Y* speichern die Koordinaten dieser Position. Dies alles erledigt Turbo Vision für Sie. Aber ein solches Objekt gehört zur Klasse der *View*-Objekte: Sie müssen die *Draw*-Methode überschreiben, um zu gewährleisten, daß der Ausschnitt aus dem Datenbereich korrekt dargestellt wird.

Mehrere Darstellungen in einem Fenster

Machen wir die Angelegenheit etwas komplizierter, indem wir dem Fenster zwei *TInterior*-Objekte zuweisen, die beide unabhängig voneinander Teile des Textes darstellen. Der Benutzer kann mittels der Maus oder über [Tab] zwischen diesen beiden Ansichten hin- und herspringen.

Um dies zu erreichen, müssen wir die Methode *MakeInterior* so ändern, daß sie feststellen kann, welcher der beiden Teile gerade aktiv ist. Das ist notwendig, weil sich die beiden Hälften etwas unterschiedlich verhalten. Weiter müssen wir in *TDemoWindow.Init* die Methode *MakeInterior* zweimal aufrufen.

```
procedure TDemoWindow.MakeInterior(Bounds: TRect; Left: Boolean);
var
  Interior: PInterior;
  R: TRect;
begin
  Bounds.Grow(-1,-1);
  Interior := New(PInterior, Init(Bounds,
                StandardScrollBar(sbHorizontal),
                StandardScrollBar(sbVertical)));
  if Left then Interior^.GrowMode := gfGrowHiY
  else Interior^.GrowMode := gfGrowHiX + gfGrowHiY;
  Insert(Interior);
end;

constructor TDemoWindow.Init(Bounds: TRect; WinTitle:
                             String; WindowNo: Word);
var
  S: string[3];
  R: TRect;
begin
  Str(WindowNo, S);
  TWindow.Init(Bounds, WinTitle + ' ' + S, wnNoNumber);
  GetExtent(Bounds);
  R.Assign(Bounds.A.X, Bounds.A.Y, Bounds.B.X div 2 + 1,
           Bounds.B.Y);
  MakeInterior(R, True);
  R.Assign(Bounds.B.X div 2, Bounds.A.Y, Bounds.B.X, Bounds.B.Y);
```

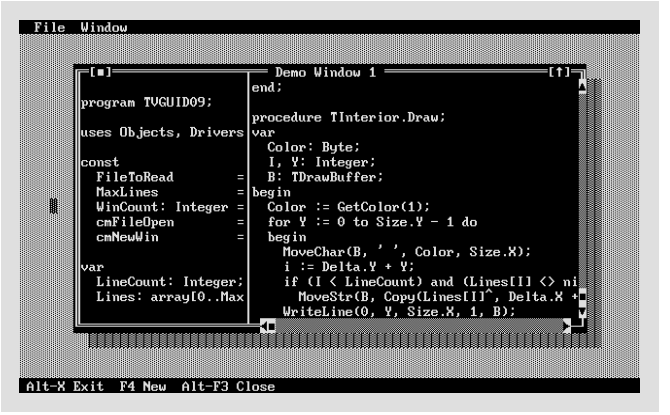


Abbildung 3.6 Das Fenster in zwei Abschnitte geteilt

```
  MakeInterior(R,False);
end;
```

Diese Programmausschnitte finden Sie in TVGUID09.PAS. Die Methode *MakeInterior* hat sowohl inhaltliche als auch stilistische Änderungen erfahren. Haben wir vorher zwei Rollbalken definiert und diese dann der *Init*-Methode übergeben, so führen wir jetzt die Aufrufe von *StandardScrollBar* als Parameter der *Init*-Methode ein. Die erste Methode ist leichter nachvollziehbar, die zweite dagegen etwas effizienter.

Wenn Sie das Fenster in TVGUID09 verkleinern, sehen Sie, daß der vertikale Rollbalken von dem linken Fensterteil überschrieben wird, sobald Sie das Fenster zu schmal werden lassen. Dieses Problem können wir überwinden, wenn wir dem Fenster eine Grenze setzen, die es in der vertikalen Ausdehnung nicht unterschreiten darf. Dazu müssen wir die Methode *TWindow.SizeLimits* überschreiben, wie in TVGUID10.PAS geschehen:

```
procedure TDemoWindow.SizeLimits(var Min, Max: TPoint);
var R: TRect;
begin
  TWindow.SizeLimits(Min, Max);
  GetExtent(R);
  Min.X := R.B.X div 2;
end;
```

Diese neue Methode müssen Sie nie selbst aufrufen. Überschreiben Sie *SizeLimits*, und automatisch wird diese neue Methode verwendet.

...und wo bringen wir die Produktion unter?

Wir haben nun ein Fenster aufgebaut, das in der Lage ist, in zwei getrennten Bereichen Text auszugeben, und jeder dieser Bereiche verfügt über zwei Rollbalken. Nehmen wir an, Sie wollten ein solches Fenster in einen voll

ausgebauten Texteditor verwandeln. Sie überlegen sich, daß Sie die Methoden, über die ein Texteditor verfügen muß, der Gruppe Bildschirmfenster zuordnen. Diesem Fenster sind schon zwei *View*-Objekte zugeordnet, die angewiesen werden könnten, den Text auszugeben. Und das ist ja auch die Aufgabe eines Fensterobjektes: ihm zugeordnete *View*-Objekte zu verwalten.

Eine Gruppe, wie sie das Fenster darstellt, ist natürlich ebenso ausbaufähig wie jedes andere Objekt auch. Jedoch werden Ihre Turbo-Vision-Programme wesentlich robuster und flexibler sein, wenn Sie einen anderen Weg verfolgen: Versehen Sie Ihre Objekte mit der größtmöglichen Autonomie. Vermeiden Sie es soweit als möglich, einen Nachkommen von *TGroup* – und *TWindow* ist ein solcher Nachkomme – mit zusätzlicher Funktionalität auszustatten.

Vermeiden Sie es also, die Fähigkeiten eines Texteditors zum Teil dem Objekt Fenster, zum Teil anderen *View*-Objekten zuzuordnen. Kreieren Sie stattdessen ein eigenes *View*-Objekt Texteditor. Wenn Sie so vorgehen, werden Sie den Texteditor auch sehr viel leichter in einem anderen Programm wieder verwenden können.

Ein Fenster schließen

In der Regel genügt ein Aufruf des Destruktors *Done*, um ein Fenster zu schließen:

```
Dispose(MyWindow, Done);
```

Die Methode *Done* ruft ihrerseits die Destruktor-Methoden aller mit dem Fenster verbundenen *View*-Objekte auf. Wenn Sie allerdings bei der Initialisierung des Fensters für irgendwelche Zwecke Speicher reservieren ließen, müssen Sie diesen Speicher innerhalb der *Done*-Methode des Fensters selbst wieder freigeben.

Ein Dialogfenster

ist eigentlich nichts anderes als ein spezialisiertes Fenster. Konsequenterweise ist das Objekt *TDialog* ein Nachkomme von *TWindow*, und wir können es tatsächlich wie ein Fenster behandeln, auch wenn es sich in einigen Punkten etwas anders verhält.

Das Programm TVGUID11.PAS beginnt mit der Ausgabe eines Dialogfensters. Eine neue Menüoption setzt den entsprechenden Befehl ab, den eine zusätzliche Methode ausführt. Eine weitere Anweisung in der *HandleEvent*-Methode schließlich verbindet den neuen Befehl mit der ebenfalls neuen Methode.

```
const
  cmNewDialog = 200;

type
  PDemoDialog = ^TDemoDialog;
  TDemoDialog = object(TDialog)
  end;

procedure TMyApp.InitMenuBar;
var R: TRect;
begin
  GetExtent(R);
  R.B.Y := R.A.Y + 1;
  MenuBar := New(PMenuBar, Init(R, NewMenu(
    NewSubMenu('~F~ile', hcNoContext, NewMenu(
      NewItem('~O~pen', 'F3', kbF3, cmFileOpen, hcNoContext,
      NewItem('~N~ew', 'F4', kbF4, cmNewWin, hcNoContext,
      NewLine(
      NewItem('E~x~it', 'Alt-X', kbAltX, cmQuit, hcNoContext,
      nil))))),
    NewSubMenu('~W~indow', hcNoContext, NewMenu(
      NewItem('~N~ext', 'F6', kbF6, cmNext, hcNoContext,
      NewItem('~Z~oom', 'F5', kbF5, cmZoom, hcNoContext,
      NewItem('~D~ialog', 'F2', kbF2, cmNewDialog, hcNoContext,
      nil)))),
    nil))
  )));
end;

procedure TMyApp.NewDialog;
var
  Dialog: PDemoDialog;
  R: TRect;
begin
  R.Assign(0, 0, 40, 13);
  R.Move(Random(39), Random(10));
  Dialog := New(PDemoDialog, Init(R, 'Demo Dialog'));
  DeskTop^.Insert(Dialog);
end;

procedure TMyApp.HandleEvent(var Event: TEvent);
begin
  TApplication.HandleEvent(Event);
  if Event.What = evCommand then
  begin
    case Event.Command of
      cmNewWin: NewWindow;
      cmNewDialog: NewDialog;
    else
      Exit;
```

```
      end;
      ClearEvent(Event);
    end;
end;
```

Nur wenige Eigenschaften unterscheiden dieses Dialogfenster von dem Fenster, das wir in den früheren Beispielen ausgeben ließen:

- Die voreingestellte Hintergrundfarbe ist grau anstatt blau.
- Das Dialogfenster kann seine Größe nicht ändern.
- Dem Fenster ist keine Nummer zugeordnet.

Um das Dialogfenster wieder zu schließen, können Sie entweder das Schließfeld anklicken, das Tastenkürzel [Alt][F3] benutzen oder [Esc] drücken. Jede dieser drei Möglichkeiten bricht den Dialog ab.

Wir haben hier Gelegenheit, einen neuen Begriff einzuführen, den wir ausführlich allerdings erst in Kapitel 5, »View-Objekte«, besprechen werden, den der *Modalität*. Unser Dialogfenster ist ein Beispiel für einen *nicht modalen* Dialog, und stellt damit einen Ausnahmefall dar. Normalerweise ist ein Dialog *modal*, und das heißt, das Programm reagiert nur auf die Ereignisse, die das modale *View*-Objekt und seine *Subviews* betreffen. Das ganze Umfeld ist sozusagen stillgelegt. Wenn Sie also die Menüleiste anklicken, wird dies keinerlei Wirkung zeigen, solange das Dialogfenster offen ist. Und dieses Verhalten ist für Dialogobjekte die Regel: es sind modale *View*-Objekte.

Einen modalen Dialog definieren

Wie aber machen Sie einen Dialog modal? Sehr einfach. Anstatt das Dialogfenster in die Gruppe Arbeitsfläche einzufügen, lassen wir den Dialog über einen Aufruf der Funktion *DeskTop^.ExecView* ausführen, wie im folgenden Ausschnitt aus TVGUID12.PAS gezeigt:

```
procedure TMyApp.NewDialog;
var
  Dialog: PDemoDialog;
  R: TRect;
  Control: Word;
begin
  R.Assign(0, 0, 40, 13);
  R.Move(Random(39), Random(10));
  Dialog := New(PDemoDialog, Init(R, 'Demo Dialog'));
  Control := DeskTop^.ExecView(Dialog);
end;
```

TDialog versteht bereits, auf die Ereignisse »Enter-Taste gedrückt« und »Esc-Taste gedrückt« zu reagieren. Das erste wird dem gerade aktuellen *TBut-*

ton zur weiteren Bearbeitung weitergereicht, das zweite in den Befehl *cmCancel* »übersetzt«. In Reaktion auf diesen Befehl wird das Dialogfenster geschlossen.

Schalter installieren

Bisher behauptet nur der Titel, unser Dialogfenster sei ein solches: Recht besehen ist es nur ein leeres Rechteck auf dem Bildschirm. Für die Anknüpfung eines Dialogs ein wenig dünn! Wir werden als erstes ein paar Schalter einbauen. Das Wort *Schalter* benutzen wir als Sammelbegriff für verschiedene Eingabeelemente in einem Dialogfenster. Sobald betätigt, setzt ein Schalter einen Befehl ab, und – das ist in diesem Zusammenhang wichtig – dieser Befehl bezieht sich ausschließlich auf Objekte, die zu dem Dialogfenster gehören.

Diese Regel kennt allerdings eine Ausnahme. Sie ist gültig dann, wenn der Dialog modal ist. Handelt es sich aber um einen nichtmodalen Dialog, gelangen diese Befehle auch an Objekte, die nicht Teil des Dialogfensters sind – und lösen dann Aktionen aus, die nicht beabsichtigt waren.

Das einfachste Schalterobjekt ist *TButton*. Es ähnelt den Einträgen in der Statuszeile: Ein andersfarbig dargestellter Bereich, versehen mit einer Beschriftung. Wenn Sie diese Beschriftung anklicken, wird ein Befehl abgesetzt. Außerdem verschwindet der Schatten unterhalb des andersfarbigen Bereichs, wodurch der Eindruck entsteht, ein Schaltknopf werde heruntergedrückt.

Dialogfenster haben wenigstens einen, meist aber zumindest zwei solcher Schalter. Einer ist mit »OK« beschriftet. Betätigen Sie ihn, dann wird der Dialog beendet und eventuell von Ihnen geänderte Werte übernommen. Der andere Schalter trägt die Aufschrift »Cancel«, der – wenn betätigt – den Dialog abbricht, wobei Änderungen verworfen werden. Dieser Schalter generiert den gleichen Befehl *cmCancel*, der auch ausgelöst wird, wenn Sie das Schließfeld des Dialogfensters anklicken.

Die Unit *Dialogs* definiert fünf Befehle, die alle mit einem *TButton*-Objekt verknüpft werden können: *cmOK*, *cmCancel*, *cmYes*, *cmNo* und *cmDefault*. Die ersten vier dieser Befehle brechen den Dialog ab, und *TDialog* ruft die Methode *EndModal* auf, die die Modalität wieder dem vor dem Dialog modalen *View*-Objekt zurückgibt.

Über diese vordefinierten Befehle hinaus können Sie in Ihren Programmen Schalter auch mit Befehlen verknüpfen, die Sie selbst festgelegt haben. Der folgende Ausschnitt aus TVGUID13.PAS zeigt, wie Sie vorgehen müssen, um Befehle an Schalter anzubinden:

```
procedure TMyApp.NewDialog;
var
  Dialog: PDemoDialog;
  R: TRect;
  Control: Word;
begin
  R.Assign(20, 6, 60, 19);
```

```
    Dialog := New(PDemoDialog, Init(R, 'Demo Dialog'));
    with Dialog^ do
    begin
      R.Assign(15, 10, 25, 12);
      Insert(New(PButton, Init(R, '~O~K', cmOK, bfDefault)));
      R.Assign(28, 10, 38, 12);
      Insert(New(PButton, Init(R, 'Cancel', cmCancel, bfNormal)));
    end;
    Control := DeskTop^.ExecView(Dialog);
  end;
```

Dem Konstruktor *Init* eines Objekts *TButton* müssen vier Parameter übergeben werden:

1. der Bereich, den der Schalter belegt (vergessen Sie nicht, daß auch der Schatten Platz belegt!);

2. die Beschriftung des Schalters;

3. der Befehl, der mit dem Schalter verbunden ist;

4. eine Angabe darüber, ob der Schalter aktiviert ist (*bfDefault*), wenn das Dialogfenster geöffnet wird, oder nicht (*bfNormal*).

Im oben gezeigten Programmausschnitt haben wir es unterlassen, dem »Cancel«-Schalter mittels des Tilde-Zeichens ein Tastenkürzel zuzuordnen, denn für diesen Schalter existiert schon das Tastenkürzel [Esc].

Normale und aktive Schalter

Über die schon erwähnten Konstanten *bfDefault* und *bfNormal* legen Sie fest, ob der betreffende Schalter aktiv ist oder nicht. Ist er aktiv, kann der Benutzer ihn über [Enter] betätigen. Natürlich kann nur immer jeweils ein Schalter in dieser Weise aktiv sein, aber Turbo Vision überprüft das nicht, darauf müssen Sie achten.

Fokussierte Schalter

Sie werden bemerken, daß in einem Dialogfenster immer einer der Schalter hervorgehoben erscheint. Er ist der *fokussierte* Schalter. Diesen Schalter kann der Benutzer »drücken«, indem er die [Leertaste] betätigt. In ein Eingabefeld kann er beispielsweise nur dann etwas eingeben, wenn dieses Objekt fokussiert ist. Mittels [Tab] kann der Benutzer den Fokus von Schalter zu Schalter bewegen, wobei bloße Beschriftungen übersprungen werden.

Aber in welcher Reihenfolge werden die Objekte mittels [Tab] fokussiert? In genau der Reihenfolge, in der die Objekte dem Dialogfenster eingefügt wurden. Diese Objekte werden in einer zirkulär verketteten Liste gehalten: Das zuletzt eingefügte Objekt ist verbunden mit dem ersten Objekt.

Um den Fokus auf ein anderes Objekt zu versetzen, rufen Sie die Methode *SelectNext* des Dialogfensters auf; Sie können aber auch die Methode *Select* des Schalters selbst benutzen. Über *SelectNext* bewegen Sie den Fokus in der Liste der Schalter vorwärts oder rückwärts. *SelectNext*(*False*) fokussiert den in der

Liste folgenden Schalter; *SelectNext*(*True*) fokussiert den in der Liste vorangehenden Schalter.

Auswahlfelder

Schalter, wie wir sie bisher kennen, erlauben nur die Aktivierung gerade einer Aktion. Aber Turbo Vision stellt auch Auswahlfelder bereit, über die der Benutzer unter mehreren Optionen auswählen kann. Die bekanntesten unter diesen Auswahlfeldern sind Markierungs- und Schaltfelder.

Diese beiden Arten von Auswahlfeldern funktionieren fast gleich, aber eben nur fast. Der Unterschied besteht darin, daß Sie über Markierungsfelder soviele der verfügbaren Optionen anwählen können, wie Sie wollen; Schaltfelder erlauben Ihnen dagegen nur die Anwahl von genau einer Option. Ein Grund für die Übereinstimmung in Erscheinungs- und Funktionsweise beider Arten von Auswahlfeldern liegt in der Abstammung, die sie gemeinsam haben: beide sind von dem Objekt *TCluster* abgeleitet.

Wenn Sie mit dem Konzept von Markierungs- und Schaltfeldern nicht vertraut sind, werfen Sie einen Blick auf die integrierte Entwicklungsumgebung von Turbo Pascal. Über das Menü *Options* öffnen Sie Dialogfenster, die solche Auswahlfelder benutzen.

Ein Auswahlfeld definieren

Ob Sie nun Markierungs- oder Schaltfelder verwenden wollen – das Vorgehen ist bei diesen beiden Arten von Auswahlfeldern genau dasselbe. Das folgende Listing – ein Ausschnitt aus TVGUID14.PAS – zeigt, was Sie tun müssen, um ein Markierungsfeld zu definieren. Dieser Ausschnitt ist Teil der neuen Methode *TMyApp.NewDialog*. Da die schon definierten Schalter alle den Dialog beenden, sollen sie die Position der als letzte in das Dialogfenster eingefügten Objekte bewahren. Dadurch bewegt sich der Benutzer mittels [Tab] zuerst über die eigentlichen Dialogelemente, bevor er zu den schon definierten Schaltern gelangt. Dazu müssen wir die folgenden Anweisungen vor der Definition der Schalter, aber nach der Initialisierung des Dialogfensters einfügen.

```
R.Assign(3, 3, 18, 6);
Bruce := New(PCheckBoxes, Init(R,
  NewSItem('~H~varti',
  NewSItem('~T~ilset',
  NewSItem('~J~arlsberg',
  nil)))
));
Insert(Bruce);
```

Die Initialisierung des Markierungsfeldes ist einfach: Wir legen ein Rechteck fest, innerhalb dessen die einzelnen Einträge ausgegeben werden, und vergessen dabei auch nicht, für die Markierungskästchen selbst genügend Platz vorzusehen. Dann bauen wir eine verkettete Liste von Zeigern auf Strings

auf, die durch **nil** abgeschlossen wird. Diese Strings dienen zur Beschriftung der Markierungsfelder.

Werte zuweisen

Als Resultat der oben zitierten Anweisungen erscheint ein Markierungsfeld, das drei Möglichkeiten zur Wahl stellt. Ob eine oder mehrere dieser Möglichkeiten schon ausgewählt sind, darüber haben wir keine Angaben gemacht; in der Voreinstellung ist keine ausgewählt. Das ist aber eher die Ausnahme. In der Regel wird die eine oder andere Option schon gewählt sein, und das Markierungsfeld soll auch zeigen, welche der Optionen ausgewählt sind.

Die Zuweisung von Ausgabewerten ist natürlich möglich, und zwar auf recht einfache Weise. Jedoch trennt Turbo Vision generell die visuelle Repräsentation von der Zuweisung und Rückgewinnung von Daten. Das bietet Ihnen den Vorteil, daß Sie ein komplettes Dialogfenster aufbauen können, ohne sich zunächst um die Daten kümmern zu müssen, die über diesen Dialog eingegeben oder verändert werden sollen. Ähnlich bei Menüs oder der Statuszeile: Auch diese können Sie darstellen lassen, ohne daß Sie die Routinen, die letztlich die zurückgegebenen Befehle ausführen sollen, schon programmiert hätten.

Auf das Problem der Bearbeitung von Daten über einen Dialog werden wir in Kürze zurückkommen. Zunächst wollen wir, damit wir die beiden Arten von Auswahlfeldern besser vergleichen können, auch ein Schaltfeld dem Dialogfenster hinzufügen:

```
R.Assign(22, 3, 34, 6);
Bruce := New(PRadioButtons, Init(R,
  NewSItem('~S~olid',
  NewSItem('~R~unny',
  NewSItem('~M~elted',
  nil)))
));
Insert(Bruce);
```

An den Anweisungen zu ihrer Initialisierung erkennen Sie schon, wie ähnlich sich diese beiden Dialogelemente sind. Der hauptsächliche Unterschied besteht darin, daß Sie über ein Markierungsfeld mehrere (oder auch keine), über ein Schaltfeld immer nur eine (aber auch mindestens eine) der angebotenen Optionen auswählen können. In der Voreinstellung gilt die erste Option als ausgewählt.

Beide Dialogelemente speichern die Optionen in einem *Word*, dessen einzelne Bits bei Markierungsfeldern darüber Auskunft geben, ob eine Option gewählt wurde oder nicht. Ein *Word* ist ein 16-Bit-Wert; daher ist auch die Zahl der in einem Markierungsfeld zur Wahl gestellten Optionen auf 16 beschränkt. Anders bei Schaltfeldern. Da über dieses Dialogelement immer nur genau eine Option gewählt werden kann, ist hier die Zahl der Schalter auf 65536 beschränkt, ein Wert, der so groß ist, daß von einer Einschränkung eigentlich nicht die Rede sein kann. Der Wert 0 gibt an, daß die erste, der Wert 1, daß die zweite Option gewählt wurde usw.

Was steht zur Wahl?

Daß dem Benutzer eines Programms die Auswahl unter verschiedenen Optionen möglich ist, wird ihm nicht genügen. Er will schließlich auch wissen, *was* er da eigentlich wählt. Turbo Vision bietet Ihnen ein weiteres Objekt, über das Sie Schaltfelder beschriften können: *TLabel*.

Das Objekt *TLabel* bietet weit mehr als es auf den ersten Blick erkennen läßt. Es gibt nicht allein Text aus, es wird automatisch auch mit einem anderen *View*-Objekt verbunden. Klicken Sie die Beschriftung an, wird das verbundene Objekt fokussiert. Außerdem können Sie auch ein Tastenkürzel definieren, indem Sie den entsprechenden Buchstaben der Beschriftung in Tilde-Zeichen (~) einschließen.

Beschriften wir also unser Markierungsfeld, indem wir die folgenden Zeilen hinter den Anweisungen, über die das Markierungsfeld selbst initialisiert wird, einfügen:

```
R.Assign(2, 2, 10, 3);
Insert(New(PLabel, Init(R, 'Cheeses', Bruce)));
```

Nun kann der Benutzer unser Markierungsfeld aktivieren, indem er dessen Beschriftung anklickt. Und er weiß darüber hinaus, was ihm zur Wahl vorgesetzt wird: *Käse*. Aber nicht nur die Käsesorte selbst, auch deren Konsistenz kann er sich aussuchen. Folgende Anweisungen beschriften das Schaltfeld:

```
R.Assign(21, 2, 33, 3);
Insert(New(PLabel, Init(R, 'Consistency', Bruce)));
```

Das Objekt TInputLine

Ein weiteres, recht einfaches Dialogelement bietet das Objekt *TInputLine*, das die Eingabe oder Änderung von Zeichenketten erlaubt. Einfach ist dieses

Abbildung 3.7
Ein voll ausgestattetes Dialogfenster

Objekt für Sie als Programmierer, da Sie es auf einfache Weise verwenden können. Intern ist es allerdings ein höchst komplexes Objekt. In TVGUID15.PAS bauen wir ein solches Eingabefeld in unser Dialogfenster ein. Die folgenden Zeilen zeigen die notwendigen Anweisungen:

```
R.Assign(3, 8, 37, 9);
Bruce := New(PInputLine, Init(R, 128));
Insert(Bruce);
R.Assign(2, 7, 24, 8);
Insert(New(PLabel, Init(R, 'Delivery instructions', Bruce)));
```

Sie sehen, ein Eingabefeld ist einfach zu definieren. Sie legen ein Rechteck fest, das die Länge des Strings auf dem Bildschirm bestimmt. Der einzig andere notwendige Parameter gibt die maximale Länge an, die die Zeichenkette erreichen darf. Diese Länge kann ohne weiteres die Breite des auf dem Bildschirm zur Verfügung stehenden Raums überschreiten; *TInputLine* ist in der Lage, bei Bedarf den Eingabestring horizontal zu rollen. Darüber hinaus kann es selbständig auf Editiertasten und Mausereignisse reagieren.

Übrigens haben wir das Eingabefeld auch gleich beschriftet. Denn mehr noch als bei Schaltfeldern ist bei Eingabefeldern eine Information darüber notwendig, welche Eingabe das Programm hier eigentlich erwartet.

Daten abliefern – Daten zurückgeben

Nun haben wir schon ein recht komplexes Dialogfenster aufgebaut. Aber rechte Freude will darüber noch nicht aufkommen. Schließlich besteht der Zweck eines solchen Fensters doch darin, Daten eingeben und Einstellungen verändern zu können. Wir haben also noch zu besprechen, wie über einen Dialog Informationen ausgetauscht werden.

Ein solcher Informationsaustausch setzt zwei Dinge voraus: Wir müssen in der Lage sein, den Dialogelementen bestimmte Werte zuzuweisen, wenn das Fenster geöffnet wird. Und wir müssen die eventuell geänderten Daten wieder zurückerhalten, sobald der Benutzer das Fenster schließt. Während das Dialogfenster geöffnet ist, verändern wir keine Daten außerhalb des Dialogs. Denn der Benutzer könnte sich ja dazu entschließen, den Dialog abzubrechen. Und dann müssen die Daten die gleichen Werte wie vor dem Dialog aufweisen. Das aber würde erschwert, hätten wir die betreffenden Daten schon während des Dialogs geändert.

Überlassen wir also Turbo Vision, Daten entgegenzunehmen und Daten zurückzugeben. Dazu übergibt Ihr Programm das Bündel relevanter Informationen an den Dialog, sobald das Fenster geöffnet wird. Wenn der Benutzer den Dialog beendet, überprüft Ihr Programm zunächst, ob der Dialog abgebrochen oder normal beendet wurde. Im ersten Fall kann das Programm mit seiner Arbeit fortfahren als sei nichts geschehen. Wurde der Dialog allerdings normal beendet, muß Ihr Programm die eventuell geänderten Daten wieder von dem Dialog zurückfordern. Dieser Informationsaustausch vollzieht sich

über die Methoden *SetData* und *GetData*. Jedes *View*-Objekt verfügt sowohl über eine *SetData*- als auch eine *GetData*-Methode.

Wird eine Gruppe (zum Beispiel *TDialog*) über einen Aufruf der Methode *SetData* initialisiert, dann verteilt sie die übergebenen Daten an die zu ihr gehörenden *Subviews*, indem sie deren *SetData*-Methode aufruft.

An diesem Punkt müssen wir zur **Vorsicht** mahnen: Wenn Sie die Methode *SetData* einer Gruppe aufrufen, übergeben Sie ihr einen Datenrecord, der die Daten für alle *View*-Objekte der Gruppe enthält. Die Daten müssen in diesem Record genau in der Folge angeordnet sein, in der die zuständigen *View*-Objekte der Gruppe hinzugefügt wurden.

Und noch einmal **Vorsicht:** Die übergebenen Daten müssen genau die Größe aufweisen, die das zuständige Dialogelement fordert. Jedes *View*-Objekt verfügt über die Methode *DataSize*, das diese Größe nennt. Und jedes *View*-Objekt kopiert genau *DataSize* Bytes aus dem Datenrecord in den eigenen Datenbereich und setzt dann einen Zeiger um den Betrag *DataSize* weiter, wodurch das nächste Objekt erfährt, an welchem Punkt des Datenrecords es seinerseits Daten übernehmen soll.

Wenn Sie ein neues *View*-Objekt ableiten und diesem Objekt weitere Datenfelder hinzufügen, müssen Sie daher auch die Methoden *DataSize*, *SetData* und *GetData* überschreiben, damit diese neuen Felder korrekt behandelt werden. All dies liegt ganz in Ihrer Verantwortung. Der Compiler kann Sie auf eventuelle Fehler oder Irrtümer nicht hinweisen.

Zeigen wir das notwendige Vorgehen für die Übergabe von Daten an einem Dialog in unserem Beispielprogramm. Zunächst machen wir Inventur im Dialogfenster: es enthält ein Markierungsfeld plus Beschriftung, ein Schaltfeld plus Beschriftung, ein Eingabefeld (für einen String von 128 Zeichen Länge) plus Beschriftung und zwei Schalter. Die Reihenfolge der Aufzählung entspricht der Reihenfolge, in der wir diese Objekte in die Gruppe eingefügt haben, und diese Reihenfolge müssen wir in dem Datenrecord einhalten, den wir zu definieren haben.

Einigen Dialogelementen brauchen wir keine Daten zu übergeben. Solche Objekte benutzen die von *TView* geerbte Methode *GetData*, die gar nichts tut. Wir können damit die Objekte *TLabel* (die Beschriftungen) und *TButton* (die Schalter) einfach unterschlagen. Bleiben noch das Markierungsfeld und das Schaltfeld (die beide, wie schon erwähnt, die Optionen in einem *Word* speichern), schließlich auch noch das Eingabefeld für einen 128 Zeichen langen String. Damit haben wir die notwendigen Angaben für die Definition des Datenrecords:

```
DialogData = record
  CheckBoxData: Word;
  RadioButtonData: Word;
  InputLineData: string[128];
end;
```

Nun müssen wir in unserem Programm eine Variable dieses Typs deklarieren:

```
var
  DemoDialogData: DialogData;
```

Diese Variable muß zu Beginn des Programms mit den gewünschten Anfangswerten belegt werden. In TVGUID17.PAS erledigen wir das innerhalb der Methode *MyApp.Init*:

```
with DemoDialogData do
begin
  CheckBoxData := 1;
  RadioButtonData := 2;
  InputLineData := 'Phone home.';
end;
```

Diesen Datenrecord übergeben wir dem Dialog bevor das Fenster geöffnet wird und überprüfen, sobald der Benutzer den Dialog beendet, ob der Dialog abgebrochen wurde. Ist das nicht der Fall, übernehmen wir die eventuell geänderten Daten:

```
Dialog^.SetData(DemoDialogData);
Control := DeskTop^.ExecView(Dialog);
if Control <> cmCancel then Dialog^.GetData(DemoDialogData);
```

Wenn Sie jetzt innerhalb des Dialogfensters irgendwelche Einstellungen verändern, werden diese Veränderungen übernommen. Das können Sie überprüfen, wenn Sie das Dialogfenster ein weiteres Mal öffnen.

Vielleicht erscheint Ihnen das Vorgehen, wie wir es eben besprochen haben, ein wenig kompliziert, und der Zwang, einen Datenrecord für die Übergabe der Daten an einen Dialog definieren zu müssen, mag Ihnen nicht recht schmecken. Aber dieser Zwang ist durchaus sinnvoll: Schließlich werden Sie in einem Dialogfenster nur solche Daten zur Veränderung anbieten, die auch etwas miteinander zu tun haben. Und warum sollte man inhaltlich zusammengehörige Daten nicht auch formal in einem Record zusammenfassen? Fördert das nicht die Durchsichtigkeit und Klarheit Ihrer Programme? Und was die Kompliziertheit angeht: Wesentlich komplizierter wäre es, wenn Sie die Daten aus allen möglichen Ecken Ihres Programms zusammensuchen müßten, bevor das Dialogfenster geöffnet wird, um sie danach auf ebenso umständliche Weise wieder an die verschiedenen Adressaten zu verteilen.

Tastenkürzel und ihre Tücken

Beschriftungen, Markierkungs- und Schaltfelder reagieren auf Tastenkürzel auch dann, wenn ein anderes Objekt innerhalb des Dialogfensters fokussiert ist. In unserem Programm zum Beispiel liegt der Fokus, wenn das Dialogfenster zum ersten Mal geöffnet wird, auf dem Markierungsfeld, und der Cursor befindet sich auf der ersten Option dieses Feldes. Drücken Sie [M] für »Melted«, so wird das Schaltfeld fokussiert, der Cursor springt auf den Auswahlpunkt *Melted*, gleichzeitig wird diese Option als ausgewählt markiert.

Ein Tastenkürzel sollte so eingängig wie möglich sein, aber dieser Forderung sind doch recht enge Grenzen gesetzt, da nur eine beschränkte Zahl von Buchstaben und Ziffern für diesen Zweck zur Verfügung stehen. In der Regel wird es immer wieder zu Konflikten kommen. In unserem Beispielprogramm beispielsweise wäre es sinnvoll [C] als Tastenkürzel für »Cheeses«, aber auch für »Consistency« festzulegen. Die Situation verschärft sich weiter, wenn wir eine neue Käsesorte, sagen wir »Cheddar«, zur Wahl anbieten. Es gibt einige Möglichkeiten, solche Probleme zu umgehen.

Wünschenswert wäre, den ersten Buchstaben eines Wortes als Tastenkürzel zu definieren. Wünschenswert, aber leider nicht immer möglich, wie wir an »Cheeses« und »Consistency« gesehen haben. Wir lösen diesen Konflikt, wenn wir uns entschließen, [O] als Tastenkürzel für »Consistency« zu verwenden. Leider ist aber dieses Tastenkürzel nicht so eingängig. Eine weitere Möglichkeit wäre, die Beschriftung zu ändern: Statt »Cheeses« verwenden wir »Kind of Cheese« und [K] als Tastenkürzel.

Derartige Manipulationen sind die einzige Möglichkeit, Konflikte zwischen Tastenkürzeln innerhalb des gleichen Gültigkeitbereiches aufzulösen. Entspringt aber ein solcher Konflikt der Gleichartigkeit von Tastenkürzeln sagen wir von Beschriftungen und einer Option eines Schaltfeldes, dann gibt es eine andere Möglichkeit, diesen Konflikt zu lösen. Tastenkürzel können als innerhalb eines Dialogelementes lokal gültig definiert werden. Nehmen wir wieder unser Beispiel und setzen voraus, wir hätten die Tastenkürzel als nur lokal gültig definiert, und der Fokus liege auf dem Markierungsfeld. Drücken Sie jetzt [M] für »Melted«, und es passiert gar nichts. Weder wird das Schaltfeld »Consistency« fokussiert, noch die Option angewählt. Das Tastenkürzel funktioniert erst dann, wenn Sie mittels [Tab] oder der Maus den Fokus auf das »Consistency«-Schaltfeld versetzen.

Voreingestellt ist, daß Tastenkürzel über das gesamte Dialogfenster hinweg gültig sind. Wollen Sie Tastenkürzel als lokal gültig definieren, müssen Sie das Feld *Options* des betreffenden Objekts ändern, bevor Sie es in den Dialog einfügen. In Kapitel 5, »View-Objekte«, behandeln wir das Feld *Options* und das Bit *ofPostProcess* ausführlich. Der folgende Programmausschnitt zeigt die notwendige Anweisung:

```
R.Assign(3, 3, 18, 6);
Bruce := New(PCheckBoxes, Init(R,
  NewSItem('~H~varti',
  NewSItem('~T~ilset',
  NewSItem('~J~arlsberg',
  nil)))
));
Bruce^.Options := Bruce^.Options and not ofPostProcess;
Insert(Bruce);
```

Nun zeigen [H], [T] und [J] erst dann eine Wirkung, wenn Sie das Markierungsfeld mittels [Tab] oder über die Maus fokussiert haben.

Denken Sie aber daran, daß ein *TLabel* niemals fokussiert werden kann. Dessen *ofPostProcess*-Bit muß also immer gesetzt sein, damit ein für dieses Objekt definiertes Tastenkürzel funktionieren kann.

Ein Nachteil dieser Methode soll hier nicht verschwiegen werden. Nur lokal gültige Tastenkürzel können den Benutzer verwirren, wenn er etwa ein solches Tastenkürzel benutzt und nichts passiert, weil gerade ein ganz anderes Dialogelement fokussiert ist. Auf der anderen Seite ist es natürlich genauso verwirrend, wenn er ein Tastenkürzel benutzt, und sich nicht die erwartete, sondern eine ganz andere Wirkung zeigt, eben weil ein Konflikt zwischen Tastenkürzeln vorliegt. Der einzige Weg, dem Benutzer solche verwirrenden Erlebnisse zu ersparen, besteht darin, ein Dialogfenster sorgfältig auszutesten, und die Definition gleicher Tastenkürzel, wenn irgend möglich, zu vermeiden.

Das Ende eines Dialogs

Wenn der Dialog abgeschlossen ist, sorgt die Methode *TGroup.ExecView*, die auch den Dialog aufgegeben hat, dafür, diesen Dialog vom Bildschirm und dann auch aus der Gruppe zu nehmen. Sie müssen nur noch über *Dispose*(*D, Done*) die Destruktor-Methode aufrufen, damit der von diesem Objekt belegte Speicher wieder freigegeben wird.

Weitere Dialogelemente

Die Unit *Dialogs* stellt weitere Objekte zur Verwendung in einem Dialogfenster bereit. Auf diese Objekte sind wir bisher nicht eingegangen, da sie auf die selbe Art verwendet werden, wie die Objekte, die wir in unsere Beispielprogramme eingebunden haben: Sie deklarieren eine Instanz des betreffenden Objekts, fügen sie in den Dialog ein und erweitern den Datenrecord um die notwendigen Daten. Wir beschränken uns im folgenden auf eine kurze Beschreibung der Funktion dieser Dialogobjekte. Für weitere Informationen müssen wir Sie auf den Referenzteil dieses Bandes verweisen.

Statischer Text

TStaticText ist ein *View*-Objekt, das einen zu übergebenden String auf dem Bildschirm ausgibt. Diese Zeichenkette wird, wenn notwendig, automatisch umbrochen, damit sie in dem zur Verfügung stehenden Rechteck dargestellt werden kann. Beginnt sie mit *Ctrl-C*, wird sie innerhalb dieses Rechtecks zentriert; Zeilenumbrüche können Sie über *Ctrl-M* erzwingen. Wie schon erwähnt, kann statischer Text nicht fokussiert werden; dieses Objekt übernimmt auch keine Daten aus dem Datenrecord.

Aus Listen wählen

Das Objekt *TListViewer* gibt in Form einer oder mehrerer Spalten eine Liste auf dem Bildschirm aus. Der Benutzer kann Einträge aus dieser Liste auswählen. Mit Hilfe zweier Rollbalken kann er den Cursor innerhalb der Liste bewegen. *TListViewer* vereinigt in sich alle notwendigen Eigenschaften zur Darstellung einer Textliste. Die Methode *GetText*, die dafür gedacht ist, die Einträge einer Liste entgegenzunehmen und an die *Draw*-Methode weiterzureichen, ist allerdings als abstrakte Methode definiert. Sie müssen also ein neues Objekt von *TListViewer* ableiten und die Methode *GetText* überschreiben, wenn Sie dieses Objekt verwenden wollen.

Einen schon verwendbaren Nachkommen von *TListViewer* stellt das Objekt *TListBox* dar. Zu ihm gehört das Objekt *TCollection*, das eine Liste von Zeigern auf Strings aufnehmen kann. *TListBox* unterstüzt nur einen Rollbalken. Innerhalb der integrierten Entwicklungsumgebung wählen Sie über ein *TListBox*-Objekt Dateien aus.

Wesentlich vereinfacht wird der Datenaustausch mit diesen Objekten über den vordefinierten Recordtyp *TListRec*, der einen Zeiger auf eine Kollektion enthält, die ihrerseits die eigentliche Textliste speichert. Darüber hinaus definiert *TListRec* ein Feld des Typs *Word*, das angibt, welcher Eintrag in der Liste selektiert ist.

Eingaben festhalten

Das Objekt *THistory* bietet eine Kombination aus einem Eingabefeld und einer Textliste. Tatsächlich ist es ein Eingabefeld, aber mit der besonderen Eigenschaft, sich an frühere Eingaben erinnern zu können. Wenn Sie das Pfeilsymbol neben dem Eingabefeld anklicken, erhalten Sie eine Auflistung der schon vorgenommenen Eingaben. Aus dieser Liste können Sie einen Eintrag auswählen, was Ihnen das wiederholte Eintippen gleicher Eingaben erspart.

Auch die integrierte Entwicklungsumgebung verwendet dieses Objekt. Sowohl über *File/Load* als auch über *Search/Find* erhalten Sie ein Dialogfenster, das ein solch komfortables Dialogelement enthält.

Standard-Dialoge

Die Unit *StdDlg* liefert das Dialog-Objekt *TFileDialog*, mit dem Sie auch in der integrierten Entwicklungsumgebung arbeiten, wenn Sie beispielsweise eine Datei laden. *TFileDialog* stützt sich auf eine Anzahl weiterer, ebenfalls in *StdDlg* definierter Objekte, die Sie sicher recht nützlich finden werden:

```
TFileInputLine = object(TInputLine)
TFileCollection = object(TSortedCollection)
TSortedListBox = object(TListBox)
TFileList = object(TSortedListBox)
TFileInfoPane = object(TView)
```

Da der gesamte Quellcode dieser Unit mitgeliefert wird, ersparen wir uns hier eine weitere Beschreibung der bereitgestellten Objekte.

4. Die Hierarchie der Objekte

Dieses Kapitel setzt voraus, daß Sie schon einige praktische Erfahrungen mit Turbo Pascal gesammelt haben, insbesondere sollten Sie dessen objektorientierte Erweiterungen gut kennen. Kapitel 1, »Objektorientiert programmieren« dieses Bandes stellt diese Erweiterungen vor. Wir setzen auch voraus, daß Sie aus der Lektüre von Kapitel 2, »Das Rad neu erfinden?«, und Kapitel 3, »Mit Turbo Vision programmieren«, einen Überblick über die Fähigkeiten, die Terminologie und das Konzept von Turbo Vision gewonnen haben.

Nach einigen allgemeinen Bemerkungen zur objektorientierten Programmierung und der Konzeption von Objekthierarchien, zeigen wir, wie die Objekte, die Turbo Vision definiert, kraft ihrer Abstammung miteinander verbunden sind. Wenn Sie die grundlegenden Eigenschaften eines Objekts

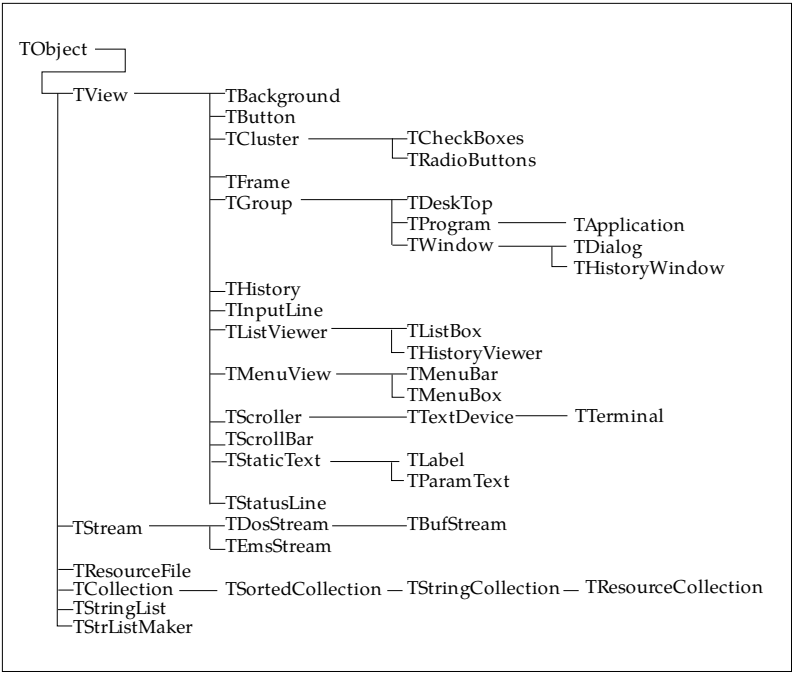


Abbildung 4.1 Die Hierarchie der Objekte in Turbo Vision

kennen, werden Sie auch bald ein Gefühl dafür entwickeln, wie ererbte und neu definierte Felder und Methoden eines jeden Objekts sich wechselseitig ergänzen, und so die Besonderheit und die Funktionalität eines speziellen Objekts ausmachen.

Den kompletten Stammbaum der in Turbo Vision definierten Standardobjekte zeigt Abbildung 4.1. Auf diesen Stammbaum werden Sie wohl immer wieder zurückkommen. Wenn Sie wissen, daß beispielsweise *TDialog* von *TWindow* und dieses wieder von *TGroup* abgeleitet ist, das seinerseits von *TView* abstammt, dann wissen Sie auch, wo Sie ansetzen müssen, wenn Sie die Eigenschaften dieser »Objektfamilie« kennenlernen und einsetzen wollen. Jedes neue Objekt, auf das Sie dann treffen, hat durch seine Abstammung bereits Ihnen vertraute Eigenschaften. Näher beschäftigen müssen Sie sich nur noch mit den neu hinzugekommenen Feldern und Methoden, über die dieses Objekt zusätzliche Eigenschaften erwirbt, die es von seinen Vorfahren abheben.

Eine Voraussetzung bei der Entwicklung von Programmen, die auf Turbo Vision aufbauen, ist eine allgemeine Vertrautheit mit den bereitgestellten Objekten und ihren wechselseitigen Abhängigkeiten. Mit den Einzelheiten werden Sie sich bald genug herumschlagen müssen, aber zunächst geht es darum, welche Objekte Sie in Ihren Programmen unmittelbar einsetzen können.

Eine »natürliche«, perfekte, sich geradzu aufzwingende Hierarchie der Objekte gibt es leider nicht. Zu entscheiden, welchem Objekt welche Eigenschaften zuzuordnen sind, erfordert sehr viel Erfahrung und ein gerüttelt Maß an Intuition und daher ist jede Objekthierarchie letztlich immer ein Kompromiß. Mit Turbo Vision profitieren Sie von unserer Erfahrung beim Entwurf einer solchen Objekthierarchie. Aber natürlich steht es Ihnen frei, Ihre eigenen Basis-Objekte zu entwerfen und so spezielle Effekte zu realisieren, die über die in Turbo Vision gebotenen hinausgehen.

Kapitel 14, »Referenz der Objekte«, beschreibt im Detail die Felder und Methoden der einzelnen Objekte. Aber wenn Sie sich noch keinen Überblick über den Stammbaum der Turbo-Vision-Objekte verschafft haben, wird Sie die Masse der dort ausgebreiteten Informationen überwältigen. Dieses Kapitel soll Ihnen diesen notwendigen Überblick verschaffen, bevor wir dann in den weiteren Kapiteln dieses Bandes auf die Einzelheiten eingehen.

Typologie der Objekte

Nicht alle Objekte in Turbo Vision haben den gleichen Status. Entsprechend ihrer Funktion lassen sich vielmehr drei Gruppen unterscheiden: Basisobjekte, *View*-Objekte und stumme Objekte. Jeder Gruppe ist in diesem Kapitel ein eigener Abschnitt gewidmet.

Aber auch die Objekte innerhalb dieser Gruppen unterscheiden sich. Von einigen können Sie direkt Instanzen deklarieren, andere wieder sind eher abstrakte Objekttypen, dazu gedacht, unmittelbar verwendungsfähige Objek-

te davon abzuleiten. Betrachten wir zunächst einmal solche abstrakten Objekte etwas näher.

Abstrakte Objekte

Viele Objekte wurden nur deshalb definiert, um von ihnen spezialisierte und unmittelbar einsatzfähige Objekte abzuleiten. Der Grund dafür mag manchmal in der Konzeption der Objekthierarchie liegen, meist aber erleichtert ein solches abstraktes Objekt die Programmierung, da dadurch die mehrfache Kodierung im wesentlichen gleicher Funktionen vermieden wird.

Nehmen wir die Objekte TRadioButtons und *TCheckBoxes* als Beispiel. Beide könnten ohne Schwierigkeiten auch direkt von dem Objekt *TView* abgeleitet werden. Dagegen spricht, daß sie viele Eigenschaften gemeinsam haben. Beides sind Auswahlfelder, die im wesentlichen gleich bedient werden. Das Schaltfeld, das ein *TRadioButtons*-Objekt zur Verfügung stellt, sieht einem Markierungsfeld, das ein *TCheckBox*-Objekt zu definieren erlaubt, sehr ähnlich. Der Unterschied liegt nur darin, daß in einem Schaltfeld immer nur eine Option, in einem Markierungsfeld dagegen auch jede Option gewählt werden kann. Die Gemeinsamkeiten sind Grund genug, ein abstraktes Objekt *TCluster* zu definieren, der beiden Objekten als gemeinsamer Vorfahre dient. Einige wenige zusätzliche Methoden genügen dann, beiden Objekten ihre individuelle Funktionalität zu verleihen.

Ein abstraktes Objekt zu instantiieren, bringt keinen großen Nutzen. Eine Instanz von TCluster, nennen wir sie *MyCluster*, besitzt keine *Draw*-Methode, sondern benutzt die von *TView* geerbte Methode *Draw*. Daher würde *MyCluster* sich einfach als leeres Rechteck auf dem Bildschirm darstellen. Wenn Sie andere Auswahlelemente definieren wollen, als Sie Ihnen *TRadioButtons* oder *TCheckBoxes* bieten, können Sie ein neues Objekt von *TCluster* ableiten. Einfacher noch mag es sein, Ihr Objekt von *TRadioButtons* oder *TCheckBoxes* abzuleiten. Welchen Vorfahren Sie auch immer für Ihr Objekt wählen, Sie werden diesem einige neue Felder und Methoden hinzufügen oder geerbte Methoden überschreiben. Haben Sie vor, eine ganze neue Familie von Auswahlelementen zu gründen, dann ist es vielleicht ganz sinnvoll, wenn Sie einen neuen abstrakten Objekttyp definieren, auf den sich diese Familie als Ahnherr bezieht.

Instanzen definieren oder Objekte ableiten

Auf jeden Objekttyp können Sie zwei Operationen anwenden. Sie können eine Instanz dieses Objekts definieren (es also *instantiieren*) oder Sie können ein weiteres Objekt von ihm ableiten. In diesem Fall haben Sie einen neuen Objekttyp, auf den Sie wiederum die beiden oben genannten Operationen anwenden können.

Instanzen definieren

Eine Instanz eines gegebenen Objekts definieren Sie, indem Sie eine statische oder dynamische Variable deklarieren:

```
MeinRollbalken: TScrollBar;
EinSchalter: PButton;
```

MeinRollbalken wird durch die Methode *TScrollBar.Init* initialisiert, wobei einige Felder mit Werten belegt werden. Genaueres dazu finden Sie in Kapitel 14, »Referenz der Objekte«, im Eintrag *TScrollBar* unter *Init*. Da *TScrollBar* ein Nachkomme von *TView* ist, ruft *Init* nun die Methode *TView.Init* auf, die ihrerseits weitere Felder des Objekts initialisiert. Der Vorfahre von *TView* ist *TObjekt*, und so wird auch die Konstruktor-Methode dieses Objekts aufgerufen, die den notwendigen Speicher reserviert. *TObject* selbst hat keinen Vorfahren, daher endet die Initialisierungskette mit diesem Objekt.

Nun sind die Felder des Objekts *MeinRollbalken* mit den voreingestellten Werten belegt, von denen Sie einige vielleicht ändern wollen. Dieses Objekt verfügt auch über alle in *TScrollBar* definierten Methoden plus der möglicherweise überschriebenen Methoden von *TView* und *TObject*. Um nun *MeinRollbalken* verwenden zu können, müssen Sie wissen, was die einzelnen Methoden tun, und das gilt insbesondere von den Methoden *HandleEvent* und *Draw*. Genügen die Eigenschaften des Objekts Ihren Anforderungen nicht, müssen Sie einen neuen Objekttyp ableiten.

Objekte ableiten

Von den Formalitäten her betrachtet ist die Ableitung eines neuen Objekttyps von einem existierenden einfach:

```
PNeuerRollbalken = ^TNeuerRollbalken;
TNeuerRollBalken = object(TScrollBar)
  end;
```

Damit haben Sie zwar einen neuen Objekttyp definiert, aber dieser unterscheidet sich von einem *TScrollBar*-Objekt noch nicht. Das erreichen Sie, wenn Sie dem Objekt *TNeuerRollbalken* neue Felder und neue Methoden hinzufügen oder auch geerbte Methoden überschreiben. Dadurch versehen Sie dieses Objekt mit neuen Eigenschaften, und diese Eigenschaften unterscheiden es von seinen Vorfahren. Definieren Sie zum Beispiel eine neue *Init*-Methode, dann können Sie die Felder dieses Objekts mit den Werten belegen, die Sie benötigen.

Die Methoden der Objekte

Die in Turbo Vision definierten Methoden lassen sich in vier, wenn auch nicht immer klar voneinander abgrenzbare Gruppen einteilen.

Abstrakte Methoden

Die abstrakten Methoden von Basisobjekten bestehen nur aus **begin**- und **end**-Anweisungen, die manchmal einen Aufruf der Prozedur *Abstract*, die einen Laufzeitfehler auslöst, umschließen. Abgeleitete Objekttypen *müssen* die abstrakten Methoden überschreiben, bevor sie benutzt werden können. Abstrakte Methoden sind daher immer auch virtuelle Methoden. Die Methode *TStream.Read* ist eine solche abstrakte Methode.

Pseudo-abstrakte Methoden

Die pseudo-abstrakten Methoden in Basisobjekten beschränken sich darauf, nur das absolut Notwendige zu tun. Meist ist es daher notwendig, diese Methoden zu überschreiben (auch sie sind virtuelle Methoden), dies ist jedoch kein Muß. Die Methode *TSortedCollection.Read* ist eine solche pseudo-abstrakte Methode.

Virtuelle Methoden

Diese Methoden erkennen Sie daran, daß ihre Deklaration mit dem reservierten Wort **virtual** abschließt. Ein abgeleitetes Objekt kann eine solche Methode überschreiben. Die überschreibende Methode muß ihrerseits auch wieder als virtuell deklariert werden, und der Kopf dieser Methode muß identisch sein mit dem Kopf der überschriebenen Methode. Virtuelle Methoden müssen nicht überschrieben werden, sind aber dafür vorgesehen, früher oder später in einem abgeleiteten Objekt überschrieben zu werden. Die Methode *TView.DataSize* ist eine solche virtuelle Methode.

Statische Methoden

Eine statische Methode kann nicht überschrieben werden. Jedoch kann ein abgeleitetes Objekt eine Methode gleichen Namens definieren, die aber möglicherweise völlig andere Parameter übernimmt, als der Namensvetter des

Vorfahren. Das hat jedoch nichts mehr zu tun mit der bei der objektorientierten Programmierung unterstützten Polymorphie und ist darüber hinaus auch gefährlich, wenn Sie die Methoden von dynamischen Objekten aufrufen. Nehmen wir an, *PGeneric* sei ein Zeiger des Typs *PView*. Mit diesem Zeiger können Sie ein beliebiges von *TView* abgeleitetes Objekt referenzieren. Wenn Sie dann aber über diesen Zeiger eine Methode dieses Objekts aufrufen wollen, wird immer die in *TView* definierte Methode angesprochen, da zum Zeitpunkt der Compilierung feststeht, daß der Zeiger vom Typ *PView* ist. Mit anderen Worten, *PGeneric^.StaticMethod* ist *immer* äquivalent zu *TView.StaticMethod*, auch dann, wenn Sie *PGeneric* einen anderen Typ zugewiesen haben. Ein Beispiel für eine statische Methode ist *TView.Init*.

Die Felder der Objekte

Wenn Sie den Mechanismus der Vererbung in der Praxis sehen wollen, dann betrachten (und vergleichen) Sie einmal die Felder der Objekte *TView*, *TGroup* und *TWindow*. Diese Objekte spielen eine wichtige Rolle in Turbo Vision.

Felder von *TView*	Felder von *TGroup*	Felder von *TWindow*
Owner	Owner	Owner
Next	Next	Next
Origin	Origin	Origin
Size	Size	Size
Cursor	Cursor	Cursor
GrowMode	GrowMode	GrowMode
DragMode	DragMode	DragMode
HelpCtx	HelpCtx	HelpCtx
State	State	State
Options	Options	Options
EventMask	EventMask	EventMask
	Buffer	Buffer
	Phase	Phase
	Current	Current
	Last	Last
		Flags
		Title
		Number
		ZoomRect
		Palette
		Frame

Das Objekt *TGroup* erbt alle Felder, die auch *TView* besitzt, und definiert einige neue, die für die Funktion dieses Objekts entscheidend sind, zum Beispiel einen Zeiger auf das aktuelle und einen Zeiger auf das letzte *View*-Objekt der Gruppe. *TWindow* wiederum erbt alle Felder von *TGroup* und fügt weitere Felder hinzu, ein Feld für die Nummer des Fensters beispielsweise und ein Feld für den Fenstertitel.

Noch einmal: Objekte ableiten

Was wir vorhin ganz allgemein (und recht kurz) gestreift haben, darauf muß jetzt etwas konkreter eingegangen werden. Einen neuen Objekttyp von einem bestehenden abzuleiten, ist formal gesehen eine einfache Sache. Aber wie nutzt dieses neue Objekt die geerbeten Fähigkeiten am effektivsten? Das Problem stellt sich gerade bei überschriebenen Methoden: wie ist deren Code zu nutzen?

Manchmal ist es notwendig, daß Ihr neues Objekt eine geerbte Methode nicht verwenden kann und sie daher überschreibt, ohne daß die neue Methode die überschriebene aufruft.

Nehmen wir als Beispiel das Objekt *TInputLine*, ein Nachkomme von *TView*. Dieses Objekt ruft die geerbte *Draw*-Methode nicht auf, denn *TView.Draw* gibt nur ein leeres Rechteck auf dem Bildschirm aus. Würde *TInputLine* diese Methode aufrufen, wäre als Resultat ein unangenehmes Flackern auf dem Bildschirm zu sehen. Daher definiert *TInputLine* eine völlig neue *Draw*-Methode:

```
procedure TInputLine.Draw;
...
begin
  { Hier finden sich die Anweisungen
    zur Darstellung des Eingabefeldes }
end;
```

Aber Methoden wie *Draw* sind eher die Ausnahme. Wenn Sie die von Turbo Vision gebotenen Möglichkeiten ausnützen wollen, dann werden in Ihrem Programm sehr viele Aufrufe geerbter Methoden zu finden sein. Aber bei jeder Methode, die Sie überschreiben, stellt sich die Frage: Rufen Sie zuerst die geerbte Methode auf und lassen dann Ihre Anweisungen ausführen? Oder erst diese Anweisungen und dann die geerbte Methode? Abhängig davon, welche Methoden Sie überschreiben, lassen sich Regeln aufstellen.

1. Sie überschreiben eine Konstruktor-Methode. Rufen Sie dann die geerbte Methode vor weiteren Anweisungen auf.

2. Sie überschreiben eine Destruktor-Methode. Lassen Sie zuerst Ihre Anweisungen ausführen und rufen Sie dann die geerbte Methode auf.

Konstruktor-Methoden überschreiben. Die Regel...

Wenn Sie eine solche Methode überschreiben, sei es nun ein *Init*- oder ein *Load*-Konstruktor, dann müssen Sie im allgemeinen zuerst den Konstruktor des Vorfahren ausführen, bevor Sie zum Beispiel den neuen Feldern Ihres Objekts Werte zuweisen:

```
type
  MyObject = object(TWindow)
    Value: Word;
    Ok: Boolean;
    constructor Init(var Bounds: TRect; ATitle: TTitleStr;
                     AValue: Word; AOk: Boolean;
  end;

constructor MyOject.Init(var Bounds: TRect; ATitle: TTitleStr;
                         AValue: Word; AOk: Boolean;
begin
  TWindow.Init(Bounds, ATitle, wnNoNumber);
  Value := 16;
  Ok := True;
end;
```

In diesem Fall wird zunächst die geerbte Methode *TWindow.Init* aufgerufen und dann erst den neuen Feldern, die *MyObject* definiert, Werte zugewiesen. Wären Sie in umgekehrter Reihenfolge vorgegangen, hätten Sie zu Ihrer Überraschung feststellen müssen, daß *Value* den Wert 0 und *Ok* den Wert False enthält. Der Grund: *TWindow.Init* folgt der oben aufgestellten Regel und ruft die Konstruktor-Methode seines Vorfahren *TGroup* auf, *TGroup.Init* wiederum *TView.Init* und diese schließlich die Konstruktor-Methode von *TObject*, des Stammvaters fast aller in Turbo Vision definierter Objekte. *TObject.Init* setzt alle Felder des Objektes *MyObject* auf 0, die Felder *Value* und *Ok* eingeschlossen.

Sie brauchen also nie in Ihren *Init*- oder *Load*-Konstruktoren irgendwelche Felder auf 0 zu setzen. Das erledigt *TObject.Init* für Sie. Sie müssen aber darauf achten, zuerst die geerbte Konstruktor-Methode aufzurufen.

...und die Ausnahme

Keine Regel ohne Ausnahme. Nehmen wir solche Objekte wie TCollection oder *TStream* Auch für sie gilt die Regel, daß die Konstruktor-Methoden abgeleiteter Objekttypen zunächst den Konstruktor ihres Vorfahren aufrufen sollten, es sei denn, es gäbe einen triftigen Grund, das nicht zu tun. Und solche Gründe gibt es. Der folgende Programmausschnitt zeigt, daß ein geerbter Konstruktor eine virtuelle Methode aufruft, also eine Methode, die Sie vielleicht überschrieben haben. *TCollection.Load* stützt sich auf die virtu-

elle Methode *GetItem*, um einen Eintrag der Kollektion aus einem Stream zu lesen:

```
constructor TCollection.Load(var S: TStream);
begin
  ...
  for I := 0 to Count - 1 do AtPut(I, GetItem(S));
end;
```

Nehmen wir an, Sie hätten die Methode *GetItem* überschrieben, und Ihre neue Methode ruft die *Load*-Methode Ihres abgeleiteten Objekts auf, damit ein Feld initialisiert und dann erst *GetItem* aufgerufen wird. Dazu ist natürlich eine neue *Load*-Methode erforderlich, die den Wert des Feldes einliest und dann *TCollection.Load* aufruft, was damit endet, daß schließlich Ihre *GetItem*-Methode aufgerufen wird. Das folgende Beispiel zeigt dies noch einmal. Die dort definierte Kollektion speichert Daten, keine Objekte. Die Größe dieser Daten ist im Feld *ItemSize* gespeichert und gilt für alle Einträge in der Kollektion.

```
type
  PDataCollection = ^TDataCollection;
  TDataCollection = object(TStringCollection)
    ItemSize: Word;
    constructor Init(ALimit, ADelta, AItemSize: Integer);
    constructor Load(var S: TStream);
    procedure FreeItem(Item: Pointer); virtual;
    function GetItem(var S: TStream): Pointer; virtual;
    function KeyOf(Item: Pointer): Pointer; virtual;
    procedure PutItem(var S: TStream; Item: Pointer); virtual;
    procedure Store(var S: TStream): virtual;
  end;

constructor TDataCollection.Load(var S: TStream);
begin
  S.Read(ItemSize, SizeOf(ItemSize));
  TStringCollection.Load(S);
end;

function TDataCollection.GetItem(var S: TStream): Pointer;
var
  Item: Pointer
begin
  GetMem(Item, ItemSize);
  S.Read(Item^, ItemSize);
  GetItem := Item;
end;
```

Die Methode *GetItem* muß wissen, wie groß der zu reservierende Speicherbereich ist, und wieviele Bytes Daten aus dem Stream zu lesen sind. Die neue *Load*-Methode ermittelt diese Größe, indem Sie zunächst den Wert für *ItemSize* aus dem Stream lädt und dann erst die *Load*-Methode seines Vorfahren

aufruft. In diesem Beispiel darf also die oben aufgestellte Regel, derzufolge die Konstruktor-Methoden abgeleiteter Objekttypen zunächst den Konstruktor ihres Vorfahren aufrufen sollten, auf keinen Fall befolgt werden.

Destruktor-Methoden überschreiben

Die Aufgabe einer Destruktor-Methode ist, all das wieder rückgängig zu machen, was die Konstruktor-Methode angerichtet hat, und zwar in umgekehrter Reihenfolge. Folgen wir dieser Logik, dann muß eine Destruktor-Methode zuerst den Speicher freigeben, den der korrespondierende Konstruktor reserviert hat, und dann den geerbten Destruktor aufrufen, der weiteren Speicher freigibt. Daraus ergibt sich für Destruktoren die Regel: Lassen Sie zuerst Ihre Anweisungen ausführen und rufen Sie dann die geerbte Destruktor-Methode auf.

Andere Methoden überschreiben

Sie haben schon gesehen, daß die Methode *TInputLine.Draw* die geerbte Methode ihres Vorfahren überhaupt nicht aufruft. Täte sie dies, müßte der Aufruf vor den weiteren Anweisungen erfolgen, denn andernfalls würde *TGroup.Draw* alle Ausgaben von *TInputLine.Draw* überschreiben. Ob und wo Sie eine geerbte Methode aufrufen, hängt von der Methode selbst, aber auch davon ab, was Sie erreichen wollen. Betrachten wir dies am Beispiel einer *HandleEvent*-Methode:

```
procedure MyObject.HandleEvent(var Event: TEvent;
begin
  { Anweisungen, die das geerbte Verhalten /ndern }
  { Aufruf der geerbten HandleEvent-Methode }
  { Anweisungen, die das neue Verhalten definieren }
end;
```

Wollen Sie also das geerbte Verhalten ändern, müssen Sie die entsprechenden Anweisungen vor dem Aufruf der geerbten *HandleEvent*-Methode einfügen. Wollen Sie das Verhalten erweitern, müssen die zuständigen Anweisungen diesem Aufruf folgen.

Die *HandleEvent*-Methode des Objekts *TDialog* ist zuständig für alle Maus- und Tastaturereignisse, die Betätigung der Taste [Tab] eingeschlossen. Nehmen wir an, Ihr Dialog-Objekt sollte die Taste [Tab] ignorieren. Sie wollen also das von *TDialog* geerbte Verhalten ändern. Daher muß in Ihre *HandleEvent*-Methode die Taste [Tab] abfangen und kann dann erst die geerbte Methode aufrufen:

```
procedure TNoTabsDialog.HandleEvent(var Event: TEvent);
begin
```

```
  if (Event.What = evKeyDown) then
    if (Event.KeyCode = kbTab) or (Event.KeyCode = kbShiftTab) then
      ClearEvent(Event);
  TDialog.HandleEvent(Event);
end;
```

Damit werden alle Tastaturereignisse ausgesiebt, die mit der Taste [Tab] zusammenhängen. Die geerbte *HandleEvent*-Methode bekommt sie nie zu sehen.

Ihre *HandleEvent*-Methode kann aber auch weitere Ereignisse entgegennehmen, die von der geerbten Methode ignoriert werden. Die meisten *View*-Objekte rufen zunächst die geerbte *HandleEvent*-Methode auf und arbeiten dann weitere Ereignisse ab:

```
procedure TDialog.HandleEvent(var Event: TEvent);
begin
  TWindow.HandleEvent(Event);
  case Event.What of
    evKeyDown:
      ...
    evCommand:
      ...
  end;
end;
```

Objekte an der Basis

Turbo Vision stellt drei einfache Objekttypen zur Verfügung, die ausschließlich von anderen Objekten benutzt werden. *TPoint* und *TRect* werden von allen *View*-Objekten verwendet; auf *TObject* baut die gesamte Objekthierarchie von Turbo Vision auf.

TPoint

Das Objekt TPoint repräsentiert einen Punkt auf dem Bildschirm. Seine Felder *X* und *Y* speichern die Koordinaten dieses Punktes. Der Ursprung des zugrundeliegenden Koordinatensystems entspricht der Position (0, 0). Dieses Objekt besitzt keine Methoden, jedoch verfügen andere Objekte über Methoden, die in der Lage sind, globale Koordinaten, die sich auf die Gesamtgröße des Bildschirms beziehen, in lokale Koordinaten, denen die Größe des von einem *View*-Objekt verwalteten Bereichs zugrundeliegt, zu konvertieren.

TRect

Dieses Objekt repräsentiert ein Rechteck. Sein Feld *A* definiert dessen obere linke, sein Feld *B* die untere rechte Ecke. Diese Felder sind je ein *TPoint*-Objekt. *TRect* definiert die Methoden *Assign*, *Copy*, *Move*, *Grow*, *Intersect*, *Union*, *Contains*, *Equals* und *Empty*. Instanzen dieses Objekts werden der *Init*-Methode eines jeden *View*-Objekts übergeben, und legen den Bereich fest, den dieses Objekt verantwortlich zu verwalten hat.

TObject

Dieses Objekt ist der Ausgangspunkt der Objekthierarchie von Turbo Vision. Als abstraktes Basisobjekt hat *TObject* keinen Vorfahren, aber viele Nachkommen. Die Objekte *TPoint* und *TRect* ausgenommen, sind alle Standardobjekte, die Turbo Vision bereitstellt, von *TObject* abgeleitet. Es besitzt keine Felder, definiert aber die drei Methoden *Init*, *Free* und *Done*. Der Konstruktor *Init* ist die Basis der Konstruktor-Methoden aller abgeleiteten Objekte und ist verantwortlich für die Reservierung des für das Objekt notwendigen Speichers. *Free* gibt diesen Speicher wieder frei. *Done* ist eine abstrakte Destruktor-Methode, die von den Nachkommen überschrieben werden muß. Jedes Objekt, das die Möglichkeiten eines *Stream* nutzen möchte, *muß* mittelbar oder unmittelbar von *TObject* abgeleitet sein.

Die Nachkommen von *TObject* lassen sich in zwei Gruppen einteilen: *View-Objekte* und *stumme Objekte*.

View-Objekte

Die auf dem Bildschirm darstellbaren Objekte in Turbo Vision nennen wir *View-Objekte*, und sie sind alle von dem Objekt *TView* abgeleitet, das wiederum ein unmittelbarer Nachkomme von *TObject* ist.

View-Objekte allgemein

Als *View*-Objekt betrachten wir jedes Objekt, das sich in einem rechteckigen Bereich auf dem Bildschirm darstellen kann. Ein solches Objekt muß immer direkt oder indirekt von *TView* abgeleitet sein. Dies ist ein abstraktes Objekt, das einen rechteckigen, leeren Bereich auf dem Bildschirm repräsentiert. Der Vorfahre *TView* stellt daher sicher, daß jedes abgeleitete Objekt über diesen Bereich und dazu auch eine virtuelle *Draw*-Methode verfügt. Diese Methode muß ein abgeleitetes Objekt allerdings überschreiben, um sich auf dem Bildschirm ausgeben zu können.

Ihre Turbo-Vision-Programme werden meistens die spezialisierteren, von

TView abgeleiteten Objekte verwenden. Aber alles, was an Turbo Vision auf dem Bildschirm sichtbar wird, geht letztlich auf *TView* zurück. Daher müssen Sie die Eigenschaften dieses Objekts kennen.

Gruppen

Einige der von *TView* abgeleiteten sichtbaren Objekte sind auch aus einem anderen Grund wichtig. Sie ermöglichen nämlich die Gruppierung von mehreren Objekten. Daraus ergeben sich komplexe Bildschirmdarstellungen, die auszugeben einem *View*-Objekt allein nicht möglich wäre.

TGroup
Das Objekt TGroup verwaltet eine dynamische, verkettete Liste miteinader kooperierender *View*-Objekte, die alle zu einem Objekt gehören: der Gruppe. Jedes *View*-Objekt besitzt das Feld *Owner*, das auf das *TGroup*-Objekt zeigt, zu dem es gehört. Der Wert **nil** dieses Feldes zeigt an, daß das betreffende Objekt nicht Teil einer Gruppe ist. Über das Feld *Next* wird die Verbindung zum nächsten Objekt der Gruppe hergestellt. Da auch eine Gruppe ein *View*-Objekt ist, kann zu dieser Gruppe auch wieder eine Gruppe gehören.

Welche weiteren Objekte zu einer Gruppe gehören, kann sich, bedingt durch die Eingaben des Benutzers, ständig ändern. Neue Subviews oder auch ganze Gruppen werden initialisiert, der Gruppe hinzugefügt und wieder daraus entfernt. Ein *View*-Objekt mag von anderen Objekt verdeckt werden und muß sich auf die Anforderung eines anderen Objekts hin wieder auf dem Bildschirm darstellen. Alle diese Aktivitäten zu koordinieren, ist die Aufgabe der Gruppe.

TDesktop
ist für den Aufbau und die Ausgabe der Arbeitsfläche zuständig, die gewöhnlich oben von der Menüleiste und unten von der Statuszeile begrenzt wird. In der Regel gehört dieses Objekt zusammen mit *TMenuBar* und *TStatusLine* zu einer von *TApplication* verwalteten Gruppe. Das Objekt *TDesktop* bietet den Hintergrund für die Aktivitäten des Programms und ist selbst eine Gruppe, zu der sich während des Programmablaufs Fenster, Listen und andere *View*-Objekte gesellen können.

TProgram
stellt eine Anzahl virtueller Methoden für seinen Nachkommen *TApplication* bereit.

TApplication
bietet die Grundlage für Ihre Turbo-Vision-Programme. Es ist, vermittelt über *TProgram*, ein Nachkomme von *TGroup* und zu ihm gehören in der Regel die Objekte *TDesktop*, *TStatusLine* und *TMenuBar*. *TApplication* verfügt über Methoden, die diese Objekte initialisieren und in die Gruppe einfügen. Die Schlüsselmethode ist allerdings *TApplication.Run*, denn sie startet das Programm.

TWindow
gibt, unterstützt von einem mit ihm verbundenen Objekt *TFrame*, die vertrauten Bildschirmfenster aus. Solche Fenster können Sie über den Bildschirm bewegen und in ihrer Größe verändern. Dafür zuständig sind die von *TView* geerbten Methoden. Ein *TWindow*-Objekt kann zoomen und sich selbst schließen. Dies erledigen die Methoden, die das Objekt selbst definiert. Ein *TWindow*-Objekt reagiert auch auf [Tab] und [Shift][Tab] und selektiert die nächste bzw. vorige Subview innerhalb des Fensters. Weiterhin übernimmt die Methode *HandleEvent* die Bearbeitung der Schließ-, Zoom- und Größenveränderungsbefehle. Ist einem Fenster eine Nummer im Bereich 1 bis 9 zugewiesen, kann es über die Tastenkombination [Alt][Nummer] selektiert werden.

TDialog
ist ein Nachkomme von *TWindow* und bietet Dialogfenster, die es dem Benutzer erlauben, Eingaben vorzunehmen. Dialogfenster enthalten in der Regel Aktionschalter, Auswahlfelder und andere Dialogelemente. Die Methode *TWindow.ExecView* sichert den aktuellen Programmkontext, fügt ein *TDialog*-Objekt der Gruppe hinzu und macht dieses Objekt dann zum modalen Objekt. Das *TDialog*-Objekt ist dann zuständig für die Bearbeitung der vom Benutzer generierten Ereignisse, also der Maus- oder Tastatureingaben. Drückt er die Taste [Esc], wird dies als Abbruchbefehl (*cmCancel*) aufgefaßt. Drückt er die Taste [Enter], wird der Befehl *cmDefault* abgesetzt, was dazu führt, daß der voreingestellte Aktionsschalter aktiviert wird. Schließlich stellt *TWindow.ExecView* den vorher gesicherten Programmkontext wieder her.

Terminale View-Objekte

sind alle *View*-Objekte, die keine Gruppen sind. Das bedeutet, daß sie sich keine anderen *View*-Objekte »aneignen« können. Sie stehen am Ende der Kette miteinander verbundener Objekte.

TFrame
Dieses Objekt versieht Bildschirmfenster mit einem Rahmen und stellt auf diesem Rahmen die Symbole für das Schließ- und Zoomfeld dar. Dieses Objekt wird niemals separat, sondern immer zusammen mit einem *TWindow*-Objekt benutzt.

TButton
TButton stellt ein spezielles Eingabemedium zur Verfügung: einen Schalter, der einen Befehl absetzt, sobald er »gedrückt« wird. Ein solches Element stellt sich auf dem Bildschirm als beschriftetes Rechteck dar. Über diese Beschriftung kann auch ein Tastenkürzel definiert werden, das diesen Schalter aktiviert. Das gleiche erreichen Sie, wenn Sie den Schalter mit der Maus anklicken. Die Bearbeitung dieser Ereignisse übernimmt das *TDialog*-Objekt, zu dem der Aktionsschalter gehört.

TCluster

Ein *TCluster*-Objekt stellt ein Feld von Schaltern bereit, die alle auf die gleiche Weise reagieren. Dieser abstrakte Objekttyp bildet die gemeinsame Grundlage für die in Turbo Vision definierten Objekttypen *TRadioButtons* und *TCheckBoxes*. Solche Schaltfelder sind meist mit *TLabel*-Objekten verbunden, wodurch die Anwahl eines Schaltfelds über die mit ihm verbundene Beschriftung möglich wird. Die *HandleEvent*-Methode erlaubt, die dargestellten Optionen sowohl mittels der Maus als auch über die Cursortasten auszuwählen.

TRadioButtons stellt ein Auswahlfeld zur Verfügung, in dem immer nur eine Option gewählt werden kann. Die Wahl einer Option deselektiert automatisch die vorher selektierte Option. Dagegen kann eine über *TCheckBoxes* zur Wahl gestellte Option unabhängig von den anderen Optionen selektiert werden.

Menüs

TMenuView ist ein abstrakter Objekttyp und gemeinsamer Vorfahre der Objekte *TMenuBar* und *TMenuBox*, die Menüleisten und Menüfenster zur Verfügung stellen. Sie übergeben Strings, die die Optionen speichern, und über die Sie gleichzeitig Tastenkürzel definieren können, und auch die Befehle, die mit den Optionen verknüpft sind. Die *HandleEvent*-Methode ist zuständig für alle im Zusammenhang mit der Auswahl aus dem Menü eingehenden Ereignisse. Möglich ist die Auswahl über die Cursortasten, über Tastenkürzel oder mittels der Maus.

Die Optionen des Hauptmenüs gibt das Objekt *TMenuBar*, die der Untermenüs das Objekt *TMenuBox* aus.

In einem Turbo-Vision-Programm haben Sie es in der Regel nicht direkt mit solchen Menü-Objekten zu tun. Sie müssen lediglich die Methode *TApplikation.InitMenuBar* überschreiben und in dieser neuen Methode die Routinen *NewMenu*, *NewSubMenu*, *NewItem* und *NewLine* aufrufen, und Turbo Vision erledigt den Aufbau dieser Menüs.

THistory

Ein *THistory*-Objekt ist immer mit einem Objekt des Typs *TInputLine* verbunden und stellt eine Liste vorher getätigter Eingaben bereit. Der Benutzer kann einen Eintrag aus dieser Liste auswählen, was ihm die wiederholte Eingabe gleicher Werte erspart. Dieses Objekt arbeitet mit den Objekten *THistoryWindow* und *THistoryViewer* zusammen.

TInputLine

TInputLine stellt einen Zeileneditor zur Verfügung, der die gewohnten Editiertasten und die Maus unterstützt. Mittels dieser kann ein Textblock innerhalb der bearbeiteten Zeile markiert werden. Diese Zeile kann länger als der am Bildschirm zur Verfügung stehende Raum sein, da sie bei Bedarf horizontal gerollt wird.

Listen ausgeben

TListViewer ist der ideale Vorfahre solcher Objekte, deren Aufgabe darin besteht, den Inhalt von Datenlisten auszugeben. Es bietet die Möglichkeit,

ein »Fenster« in zwei Richtungen über einen wesentlich größeren Datenbereich zu bewegen. Die Darstellung der Liste kann über ein oder zwei Rollbalken gesteuert und Einträge können mittels der Maus oder über die Tastatur selektiert werden. Die Methode *Draw* kommt auch mit einer Größenänderung des Darstellungsbereichs zurecht. *TListViewer* definiert eine abstrakte Methode *GetText*, die dazu gedacht ist, dem Objekt das Einlesen der darzustellenden Daten zu ermöglichen. Sie müssen diese Methode überschreiben, damit die Daten in der von Ihnen gewünschten Form ausgegeben werden können.

TListBox ist ein von *TListViewer* abgeleitetes Objekt, das Ihnen ermöglicht, Listen miteinander verketteter Strings, zum Beispiel eine Liste von Dateinamen, auszugeben. Eine solche Liste kann in ein oder mehreren Spalten innerhalb des Darstellungsbereichs ausgegeben werden, wobei auch ein vertikaler Rollbalken, nicht jedoch ein horizontaler Rollbalken unterstützt wird. Die von *TListViewer* geerbten Methoden erlauben die Selektion eines Eintrags der Liste mittels der Maus oder über die Cursortasten. Dieses Objekt definiert das Feld *List*, das einen Zeiger auf ein *TCollection*-Objekt speichert, in dem die darzustellenden Einträge enthalten sind. Was diese Kollektion enthält, liegt in Ihrer Verantwortung, und das gilt auch für die Aktion, die ausgelöst wird, wenn ein Eintrag der Liste selektiert ist.

TSroller und TScrollBar

Das Objekt TScroller verhält sich wie ein Guckloch, durch das Sie Ausschnitte aus einem wesentlich größeren Datenbereich betrachten können. Die Position dieses Gucklochs wird verändert entweder über die Cursortasten oder mittels Anklicken der mit diesem Objekt verbundenen Rollbalken. Die Felder *HScrollId* und *VScrollId* verweisen auf den horizontalen und vertikalen Rollbalken. Das Feld *Delta* bestimmt zusammen mit entsprechenden Feldern der Rollbalken die Schrittweite, mit der dieses Objekt über den Datenbereich zu bewegen ist.

Das Objekt TScrollBar bietet einen vertikalen oder horizontalen Rollbalken. Das Feld *Value* speichert die Position des Indikatorzeichens, die Felder *PgStep* und *ArStep* bestimmen die Schrittweite, mit der die Datenansicht zu rollen ist, wenn die entsprechenden Maus- oder Tastaturereignisse eintreffen.

Diese beiden Objekte gehören normalerweise zu einem *TWindow*-Objekt, das damit oft komplexe Operationen zu verwalten hat. Denn wenn es zu Beispiel seine Größe ändert, müssen die Objekte *TScroller* und *TScrollBar* ihre Größe entsprechend anpassen.

TTextDevice und TTerminal

TTextDevice nutzt die von seinem Vorfahren *TScroller* geerbten Fähigkeiten, um ein einfaches TTY-ähnliches, textorientiertes Ein-/Ausgabegerät zu implementieren. Die virtuellen Methoden, die dieses Objekt definiert, dienen dazu, Strings einzulesen und auszugeben. *TTextDevice* stellt lediglich ein Basis-Objekt für die Ableitung von Terminal-Objekten dar.

TTerminal implementiert ein einfaches Terminal, über das Strings gelesen und geschrieben werden können. Diese Operationen laufen über einen Ringpuffer, dessen Größe bei der Initialisierung festgelegt wird.

Statischer Text

TStaticText ist das einfachste aller in Turbo Vision definierten *View*-Objekte: es beschränkt sich darauf, den über das Feld *Text* angesprochenen String auszugeben, und reagiert weder auf Tastatur- noch auf Mausereignisse.

TLabel ist ein Nachkomme von *TStaticText*, unterscheidet sich von diesem aber dadurch, daß ein Label mittels der Maus, über die Cursortasten oder auch über eine Tastenkombination aus `[Alt][Buchstabe]` selektiert werden kann. Normalerweise ist ein Label über einen *PView*-Zeiger mit einem anderen Dialogelement, einem Eingabefeld, einem Auswahlfenster oder einem Schalterfeld, verbunden. Wird das Label selektiert, wird auch das verbundene Element selektiert. Wird umgekehrt dieses Dialogelement selektiert, wird auch das Label hervorgehoben dargestellt.

TStatusLine

Dieses Objekt wird in der untersten Zeile des Bildschirms dargestellt. Sie können es zur Ausgabe von Informationen verwenden: Angaben über verfügbare Tastenkürzel oder Kurzinformationen zu Menüoptionen. Die in der Statuszeile gezeigten Optionen können über Tastenkürzel oder mittels der Maus selektiert werden. Das Objekt reagiert auf Ereignisse, die das Programm übermittelt, durch Ausgabe eventuell anderer, passender Informationen. Wie die Menüleiste und das *Desktop*-Objekt gehört auch die Statuszeile zu der Gruppe *TApplication*.

Das Feld *Items* dieses Objekt zeigt auf die aktuelle verkettete Liste von *TStatusItem*-Records. Diese speichern die auszugebenden Strings, die Tastenkürzel und die mit ihnen verknüpften Befehle. Das Feld *Defs* zeigt auf eine verkettete Liste von *PStatusDef*-Records, über die Sie zum aktuellen Programmkontext passende Kurzinformationen ausgeben können. Ein *TStatusLine*-Objekt kann über *TApplication.InitStatusLine* initialisiert werden.

Stumme Objekte

Zu dieser Gruppe von Objekten, die sich nie selbst auf dem Bildschirm bemerkbar machen, gehören Streams, Ressourcen-Dateien, Kollektionen und String-Listen.

Stream-Objekte

Traditionelle Methoden der Ein- und Ausgabe in und aus einer Datei versagen bei polymorphen Objekten. Daher definiert Turbo Vision spezielle Stream-Objekte, die genau diese Aufgabe erledigen.

TStream ist ein abstraktes Objekt, das grundlegende Methoden zur Ein- und Ausgabe polymorpher Objekte bereitstellt. Wenn Sie von diesem Objekt abgeleitete eigene *Stream*-Objekte definieren, müssen Sie die abstrakten Me-

thoden *GetPos*, *GetSize*, *Read*, *Seek*, *Truncate* und *Write* überschreiben. Die in Turbo Vision definierten und von *TStream* abgeleiteten Objekte *TDosStream* und *TEmsStream* definieren das dazu nötige Vorgehen. Wickelt Ihr *Stream*-Objekt die Ein- und Ausgabe über einen Puffer ab, müssen Sie zusätzlich auch die Methode *Flush* neu definieren.

Objekte, die in einem Stream gespeichert oder aus einem Stream gelesen werden sollen, müssen direkt oder indirekt von *TObject* abstammen und über die Prozedur *RegisterType* registriert worden sein. Das ist für die in Turbo Vision definierten Standardobjekte schon geschehen (vergleichen Sie den Eintrag »RegisterType-Prozedur« in Kapitel 15, »Außerhalb der Objekte«).

TDosStream

TDosStream ist ein spezialisierter Nachkomme von *TStream* und implementiert Ein- und Ausgabe polymorpher Objekte in bzw. aus einer DOS-Datei. Die Schlüsselmethode von *TDosStream* ist der Konstruktor *Init*, der erlaubt, je nach dem Wert seines Parameters *Mode* eine DOS-Datei zu erstellen oder eine schon vorhandene zu öffnen. Das einzige Feld, das *TDosStream* definiert, ist der Datei-Handle, über den der Zugriff auf die Datei abgewickelt wird. Dieses Objekt überschreibt alle abstrakten Methoden von *TStream*, ausgenommen dessen Methode *Flush*, die bei Ein- und Ausgaben über einen Puffer verwendet wird.

TBufStream

TBufStream ist ein Nachfolger von *TDosStream*, der im Unterschied zu diesem Ein- und Ausgaben über einen Puffer abwickelt. Die zusätzlichen Felder enthalten einen Zeiger auf diesen Puffer und geben dessen Größe an. Zwei weitere Felder speichern die aktuelle Position in diesem Puffer und zeigen seinen Füllungsgrad an. Zusätzlich zu den acht Methoden, die auch schon *TDosStream* überschreibt, definiert *TBufStream* die Methode *Flush* neu. Diese Methode schreibt den Inhalt des Puffers in die Datei, bevor sie geschlossen wird.

TEmsStream

ist ein weiterer spezialisierter Nachkomme von *TStream* und nutzt den EMS-Speicher zur Ablage polymorpher Objekte. Die zusätzlichen Felder, die *TEmsStream* definiert, speichern den EMS-Handle, führen Buch über die belegten Speicherseiten, die Größe des Stream und die aktuelle Position.

Ressourcen

TResourceFile stellt einen Stream zur Verfügung, dessen Einträge über einen Schlüssel des Typs String indiziert sind. Das Feld *Stream* verweist auf das Stream-Objekt, das *TResourceFile* benutzt. Sie können mittels der Methode *TResourceFile.Get*, der Sie den Schlüssel-String übergeben, auf einen Eintrag zugreifen. Über *TResourceFile.Put* schreiben Sie einen Eintrag in die Ressourcen-Datei. Die Methode *KeyAt* ermittelt den Index eines bestimmten Eintrags,

Flush schreibt alle Änderungen in den Stream und *Count* gibt die Anzahl der Einträge in der Datei zurück.

Kollektionen

T*CollectionTCollection* bietet eine allgemeine Grundlage zur Implementierung eines »Behälters« für Daten und Objekte. Das Konzept, das *TCollection* zugrundeliegt, ist wesentlich flexibler als das von Arrays, Sets oder Listen. Eine Kollektion paßt sich zur Laufzeit dynamisch an die Größe der ihr übergebenen Daten an und bildet die Grundlage für spezialisiertere Objekttypen, wie *TSortedCollection*, *TStringCollection* und *TResourceCollection*. Das Feld *Items* zeigt auf ein Zeiger-Array, über das die eigentlichen Einträge angesprochen werden. Das Objekt verfügt über Methoden, die erlauben, Einträge hinzufügen oder Einträge zu löschen. Darüber hinaus stehen auch Wiederholmethoden bereit, die eine vom Anwender zu definierende Funktion auf jeden Eintrag in der Kollektion anwenden.

TSortedCollection

TSortedCollection ist ein spezialisierter Nachkomme von *TCollection*, und stellt eine Kollektion bereit, deren Einträge nach einem bestimmten Schlüssel sortiert sind. Doppelte Einträge werden zurückgewiesen. Für die Sortierordnung verantwortlich ist die Methode *Compare*, eine abstrakte Methode, die Sie überschreiben müssen, um die Sortierordnung festzulegen, die Sie benötigen. Neue Elemente werden dann in der Reihenfolge eingefügt, die *Compare* bestimmt. Die Sortierschlüssel, die *Compare* vergleicht, stellt die Methode *KeyOf* bereit, die Sie ebenfalls überschreiben müssen. Ein Element können Sie über *Search* finden; die Methode führt eine binäre Suche in der Kollektion durch.

TStringCollection

TStringCollection ist ein Nachkomme von *TSortedCollection* und stellt eine Kollektion für die Speicherung von Strings bereit. Die *Compare*-Methode von *TSortedCollection* wird überschrieben, um die lexikalisch sortierte Anordnung von Zeichenketten zu gewährleisten. Die Methode *FreeItem* entfernt den anzugebenden String aus der Kollektion. Damit eine solche Kollektion mit einem Stream zusammenarbeiten kann, stehen auch die virtuellen Methoden *PutItem* und *GetItem* bereit.

TResourceCollection

TResourceCollection ist ein Nachkomme von *TStringCollection* und stellt in Zusammenarbeit mit *TResourceFile* eine Kollektion bereit, die Ressourcen speichert. Eine Ressourcen-Datei ist nichts anderes als ein *Stream*, indiziert über einen String. Einzig um dieses Schlüsselfeld richtig behandeln zu können, überschreiben die Methoden von *TResourceCollection* die entsprechenden Methoden ihrer Vorfahren.

Strings in Listen

TStringList stellt eine spezielle Ressourcen-Datei zur Verfügung, die Strings speichert, die über eine Nummer angesprochen werden. Dies erlaubt Ihnen zum Beispiel, Ihr Programm relativ einfach auf eine andere Landessprache umzustellen, da dafür nur diese Strings zu ändern sind. Das Feld *Count* speichert die Anzahl der in der Datei gespeicherten Einträge, die über die Methode *Get* gelesen werden.

TStringList definiert nur Methoden, die das Lesen dieser indizierten Strings erlauben. Benutzen Sie das Objekt TStrListMaker, um eine solche Ressourcen-Datei zu erstellen. Dieses Objekt kann über die Methode *Put* Strings in eine Liste einfügen und über die Methode *Store* diese Liste in einem Stream schreiben.

5. View-Objekte

Sie haben – das hoffen wir jedenfalls – Kapitel 1 »Objektorientiert programmieren« gelesen. Sie kennen – und das vermuten wir wohl mit Recht – die integrierte Entwicklungsumgebung von Turbo Pascal. Und daher wissen Sie auch, wie ein Turbo-Vision-Programm seinem Benutzer gegenübertritt. Dieses Design ist inzwischen den Benutzern verschiedenster Computer vertraut.

Soweit das Äußere. Doch was spielt sich hinter der Szene ab? Darauf näher einzugehen, haben wir uns für dieses und das folgende Kapitel vorgenommen.

Turbo Vision übernimmt ...

Wir haben schon des öfteren darauf hingewiesen und nur der Sicherheit wegen sei es hier noch einmal wiederholt: Wenn Sie Turbo Vision einsetzen, dann müssen Sie Turbo Vision die Verwaltung des Bildschirms überlassen. Anstatt über *Write* oder *WriteLn* eine Information auf dem Bildschirm auszugeben, reichen Sie diese Information an Turbo Vision weiter, und Turbo Vision sorgt dafür, daß sie zur richtigen Zeit an der richtigen Stelle des Bildschirms erscheint.

View-Objekte bilden das Fundament, auf dem Turbo-Vision-Programme aufbauen. Es sind Turbo-Pascal-Objekte, deren Aufgabe darin besteht, einen rechteckigen Bereich auf dem Bildschirm verantwortlich zu verwalten. Und das betrifft nicht nur die Darstellung selbst: Ein *View*-Objekt muß darüber hinaus auch in der Lage sein, auf Eingaben des Benutzers, die diesen Bereich betreffen, angemessen zu reagieren. Nehmen wir als Beispiel die Menüleiste; auch sie ist ein *View*-Objekt, und wenn Sie mit der Maus eine Option anklicken, ist es Aufgabe dieses Objekts, die Darstellung des gewählten Untermenüs zu veranlassen.

Wie Menüs sind auch Bildschirmfenster, Schalter, Rollbalken, ja selbst einfache Textzeilen *View*-Objekte. Generell gilt: Alles was sich innerhalb eines Turbo-Vision-Programms auf dem Bildschirm bemerkbar macht, muß ein *View*-Objekt sein. Die herausragende Eigenschaft eines solches Objekts besteht in seiner Fähigkeit, sich selbst auf dem Bildschirm darstellen zu können. Wenn Sie also zum Beispiel ein Menüsystem entwickeln wollen, dann teilen Sie Turbo Vision einfach mit, daß es eine Menüleiste aufbauen soll, die eine

gewisse Zahl von Untermenüs enthält, und Turbo Vision übernimmt diese Aufgabe.

Wäre eine Steigerung von »sichtbar« überhaupt möglich, würden wir sagen, das sichtbarste *View*-Objekt einer Turbo-Vision-Anwendung sei dieses Programm selbst. Dieses Objekt kontrolliert den gesamten Bildschirm. Sie merken davon nichts, denn es delegiert diese Aufgabe an andere *View*-Objekte, die wir im folgenden *Subviews* nennen wollen. Viele Objekte, die auf dem Bildschirm als ein Ganzes erscheinen – zum Beispiel ein Fenster – sind in Wirklichkeit Gruppen miteinander verknüpfter *View*-Objekte.

Einfache View-Objekte

Wie Sie der Darstellung der Objekt-Hierarchie in Abbildung 5.6 entnehmen können, haben alle *View*-Objekte einen gemeinsamen Vorfahren: *TObject*. Dieses Objekt stellt tatsächlich den Ahnherrn *aller* in Turbo Vision definierter Objekte dar. Ein *View*-Objekt ist allerdings nicht direkt, sondern indirekt über *TView* von dem Objekt *TObject* abgeleitet.

TView selbst erscheint auf dem Bildschirm als leeres Rechteck. Es gibt jedoch wenig Gründe dafür, daß Sie eine Instanz direkt von *TView* deklarieren sollten, es sei denn, Sie wollten ein in Turbo Vision noch nicht vorhandenes *View*-Objekt definieren. Obgleich sich nun *TView* auf dem Bildschirm so unscheinbar präsentiert, enthält es alle für die Verwaltung des Bildschirms durch Turbo Vision fundamentalen Methoden und Felder.

Zu was muß nun ein von *TView* abgeleitetes Objekt fähig sein? Wie schon kurz angesprochen zu zwei Dingen:

- Es muß jederzeit in der Lage sein, sich auf dem Bildschirm darzustellen. *TView* definiert eine virtuelle Methode – *Draw* – und über diese Methode muß auch jedes von *TView* abgeleitete Objekt verfügen. Das ist deshalb so wichtig, weil jedes *View*-Objekt ganz oder teilweise durch eine andere *View* verdeckt werden kann. Wird das überdeckende Objekt über den Bildschirm bewegt oder ganz entfernt, muß das darunterliegende die vorher verdeckten Teile seiner selbst wieder restaurieren können.

- Es muß auf hereinkommende Ereignisse – *Events* – angemessen reagieren können. Wir erwähnten es schon: Turbo-Vision-Anwendungen sind ereignisgesteuerte Programme. Turbo Vision nimmt die Eingaben des Benutzers entgegen und schickt sie an die zuständigen Objekte weiter. Ein *View*-Objekt muß mit diesen Ereignissen richtig umgehen können. Auf die Einzelheiten gehen wir in Kapitel 4, »Ereignisgesteuerte Programme«, näher ein.

Wo die Darstellung beginnt

Bevor wir darüber sprechen können, was *View*-Objekte tun, müssen wir

näher behandeln, wie sich solche Objekte einerseits auf dem Bildschirm darstellen und wie sie sich andererseits dort verhalten.

Die Position einer *View* auf dem Bildschirm wird bestimmt durch zwei Angaben, die obere linke und die untere rechte Ecke. Die obere linke Ecke bezeichnen wir auch als den *Ursprung*. Beide Punkte speichert das Objekt in je einem Feld des Typs *TPoint*, wobei *Origin* den Ursprung und *Size* den unteren rechten Punkt beschreibt.

Beachten Sie, daß *Origin* einen Punkt im Koordinatensystem der übergeordneten *View* darstellt. Wenn Sie zum Beispiel ein Fenster öffnen, dann beziehen sich dessen X- und Y-Koordinaten auf den Ursprung der Arbeitsfläche. Die Werte des Feld *Size* dagegen sind als relativ zum Ursprung des Objektes selbst zu verstehen, und geben somit die Distanz der unteren rechten von der oberen linken Ecke an. Wo auf dem Bildschirm die untere rechte Ecke eines *View*-Objekts sich genau befindet, können Sie also erst dann ermitteln, wenn Sie den Ursprung dieses Objekts kennen.

Der schon erwähnte Typ *TPoint* definiert nur zwei Felder, *x* und *y*, zur Aufnahme der Koordinatenwerte. Methoden besitzt dieses Objekt nicht. Turbo Vision benutzt *TPoint*, damit *View*-Objekte ihre Koordinaten in einem einzigen Feld speichern können.

Allerdings werden wir kaum jemals auf einen *TPoint* direkt zugreifen müssen. Da jede *View* einen rechteckigen Bereich verwaltet, der sowohl einen Ursprung als auch eine Ausdehnung hat, stellt Turbo Vision das Objekt *TRect* bereit, das in den beiden Feldern *A* und *B* - beide vom Typ *TPoint* – diese Werte verwaltet. Wenn Sie die Ausmaße eines *View*-Objekts festlegen, übergeben Sie diese Abmessungen in Form eines *TRect* dem zuständigen Konstruktor.

Sowohl *TRect* als auch *TView* definieren nützliche Methoden zur Änderung der Größe einer *View*. Nehmen wir an, Sie wollten ein *View*-Objekt definieren, das den Innenraum eines Fensters genau ausfüllt. Dazu erfragen Sie die Größe dieses Fensters, verringern die gelieferten Werte und übergeben Sie dem neuen *View*-Objekt:

```
procedure ThisWindow.MakeInside;
var
  R: TRect;
  Inside: PInsideView;
begin
  GetExtent(R);
  R.Grow(-1, -1);                    { um -1 "verg·øern" }
  Inside := New(PInsideView, Init(R));
  Insert(Inside);                  { dem Fenster zuordnen }
end;
```

Beide Methoden, auf die es uns hier ankommt, sind in *TView* definiert. *GetExtent* gibt über *R*, ein Objekt vom Typ *TRect*, die Koordinaten der übergeordneten *View* zurück; *Grow* vergrößert oder – wenn die übergebenen Werte negativ sind – verkleinert die horizontale bzw. vertikale Seite des Rechtecks.

Koordinatensystem

Das Koordinatensystem, das Turbo Vision zugrundelegt, ist ein wenig anders als Sie es wahrscheinlich gewohnt sind. Die Standardprozedur *GotoXY* beispielsweise erwartet als Argumente die Spalte und die Zeile, um den Cursor auf diesen Punkt zu setzen, wobei die Werte 1, 1 den Ursprung bezeichnen. Das Koordinatensystem von Turbo Vision dagegen bezieht sich nicht auf die Zeichen, sondern auf das Gitter zwischen ihnen.

Wenn wir voraussetzen, *R* sei ein Objekt des Typs *TRect*, dann beschreibt R.Assign(0,0,0,0) ein Rechteck ohne Ausdehnung, faktisch also einen Punkt. Das kleinste Rechteck, das auch etwas aufnehmen kann, würde über R.Assign(0,0,1,1) definiert werden.

In Abbildung 5.1 sehen Sie ein Rechteck, definiert über die Anweisung R.Assign(2,2,4,5).

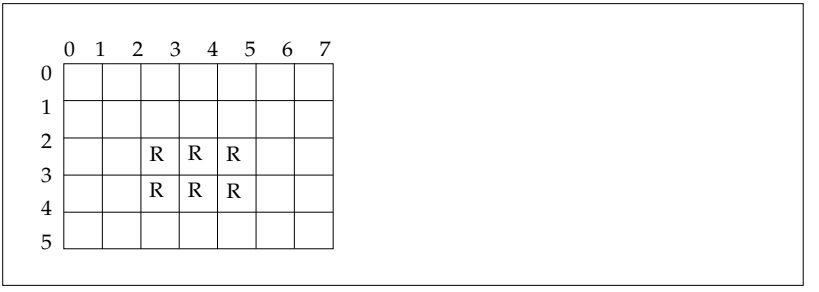


Abbildung 5.1 Das Koordinatensystem von Turbo Vision

Wie Sie sehen, erzeugt R.Assign(2,2,4,5) ein Rechteck, das Raum bietet für insgesamt sechs Zeichen. Dieses Koordinatensystem mag ein wenig unkonventionell erscheinen, bietet aber den Vorteil, die Berechnung der Größe des Rechtecks als auch angrenzender Flächen beträchtlich zu erleichtern.

Wie der Darsteller auftritt

legt die Methode *Draw* des Objekts fest. Wohl jeder neue Objekttyp muß über seine eigene *Draw*-Methode verfügen, denn es ist gerade seine Darstellung, die ein *View*-Objekt von einem anderen unterscheidet.

Eine *View*-Objekt muß mittels seiner *Draw*-Methode zwei Forderungen erfüllen:

- Es muß den ganzen Bereich überdecken, für den dieses Objekt verantwortlich ist.
- Es muß sich jederzeit auf Anforderung selbst auf dem Bildschirm darstellen können.

Auf diese beiden Anforderungen und ihre Gründe müssen wir etwas näher eingehen.

Territoriale Verantwortung

Jedem *View*-Objekt wird ein rechteckiger Bereich auf dem Bildschirm zugewiesen. Wenn es nun nicht in der Lage ist, diesen ihm zugewiesenen Bereich zu überdecken, wird dort alles möglich zu sehen sein. Was genau, ist durch den Zufall bedingt, jedenfalls entzieht sich dieser Bereich dann Ihrer Kontrolle. Das Programm TVGUID05.PAS zeigt, was passiert, wenn ein *View*-Objekt sich seiner Verantwortung für den von ihm verwalteten Bereich entzieht.

Jederzeit bereit

Darüber hinaus muß sich ein *View*-Objekt jederzeit auf dem Bildschirm darstellen können. Denn andere Objekte, die dieses Objekt ganz oder zum Teil überdecken, könnten vom Bildschirm genommen werden. Oder aber das *View*-Objekt wird über den Bildschirm bewegt. Wie auch immer: das Objekt muß in der Lage sein, sich mittels seiner Methode *Draw* auf dem Bildschirm ausgeben zu können.

Das kann unter Umständen auch bedeuten, daß das Objekt überhaupt nichts tun muß. Es kann ganz von anderen Objekten überdeckt sein oder das Fenster, innerhalb dessen es erscheint, kann so verkleinert worden sein, daß das Objekt überhaupt nicht mehr sichtbar ist. Derlei Situationen werden automatisch berücksichtigt, wichtig ist jedoch, daß das Objekt über die Fähigkeit verfügt, sich auf Anforderung selbst darzustellen.

Das ist prinzipiell verschieden von dem Vorgehen anderer fensterorientierter Systeme. Dort lassen Sie etwas in einem Fenster ausgeben, und das bleibt im Prinzip sichtbar, auch wenn es zeitweise von anderen darüberliegenden Fenstern überdeckt werden mag. In Turbo Vision können Sie nicht davon ausgehen, daß die Darstellung eines Objekts, das für einige Zeit von anderen Objekten überdeckt war, noch korrekt ist. Denn in der Zwischenzeit hat sich möglicherweise sein Status geändert.

Wie der Darsteller agiert

ist festgelegt durch die Methode *HandleEvent*. Dieser Methode wird ein Ereignisrecord übergeben, und dieses Ereignis muß sie in einer von zwei möglichen Weisen bearbeiten. Sie kann als Antwort irgendeine Aktion veranlassen und dann das Ereignis als bearbeitet markieren. Sie kann aber auch dieses Ereignis an ein anderes *View*-Objekt zur Bearbeitung weiterreichen. In Kapitel 6, »Ereignisgesteuerte Programme«, beschreiben wir näher, was Ereignisse sind, und wie Turbo Vision sie bearbeitet.

Die Art und Weise, wie ein *View*-Objekt sich verhält, ist bestimmt dadurch, wie es auf Ereignisse reagiert. Wird beispielsweise einem Fenster ein *cmClose*-Befehlsereignis übermittelt, reagiert es auf dieses Ereignis, indem es sich schließt. Aber es ist natürlich möglich, wenn auch wahrscheinlich nicht sonderlich sinnvoll, daß Sie für dieses Objekt eine andere Reaktion auf den Empfang dieses Ereignisses definieren.

Komplexe Darstellungen

Ein unmittelbarer Nachfahre von *TView*, und unter ihnen auch der wichtigste, ist *TGroup*, über den Sie schon einiges wissen. Er selbst und alle von ihm abgeleiteten Objekte bezeichnen wir allgemein als *Gruppe*. *View*-Objekte dagegen, die nicht von *TGroup* abgeleitet wurden, betrachten wir als *terminale Views*.

Im Grunde genommen ist eine Gruppe nichts weiter als eine leere Schachtel, bereit, andere *View*-Objekte aufzunehmen und zu verwalten. Technisch gesehen ist auch eine Gruppe ein *View*-Objekt, das über jene Fähigkeiten verfügen muß, die allen solchen Objekten abverlangt werden: einen Bereich des Bildschirm verantwortlich zu verwalten, sich selber darstellen zu können, auf zutreffende Ereignisse zu reagieren. Der Unterschied liegt darin, *wie* eine Gruppe diese Aufgaben erledigt: sie überläßt die eigentliche Arbeit ihren *Subviews*.

Gruppen und Subviews

Eine *Subview* ist ein Objekt, das zu einem anderen *View*-Objekt gehört, das damit zu einer Gruppe avanciert und einen Teil seiner Verantwortung auf diese *Subview* abwälzen kann: Sie ist dann verantwortlich für einen Teil des Bereichs, den das übergeordnete Objekt einnimmt.

Das augenfälligste Beispiel für eine Gruppe ist *TApplication* selbst, dessen Bereich den gesamten Bildschirm umfaßt. Aber zu der Gruppe *TApplication* gehören drei *Subviews*: die Menüleiste, der Arbeitsbereich, die Statuszeile. Jedem dieser Objekte überantwortet *TApplication* einen Teil des Bildschirms: Die Menüleiste darf die oberste Zeile und die Statuszeile die unterste Zeile übernehmen; der Arbeitsbereich verantwortet alles, was dazwischenliegt. Aus dieser Aufteilung folgt der für *TApplication* typische Bildschirmaufbau wie in

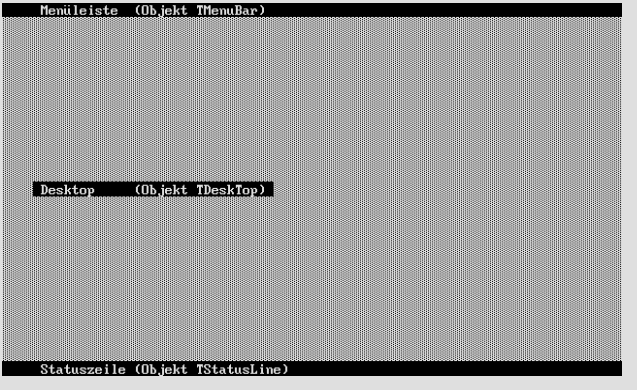


Abbildung 5.2 Bildschirmaufteilung von TApplication

Abbildung 5.2 gezeigt. Sie bemerken, daß die Gruppe *TApplication* selbst nicht auf dem Bildschirm erscheint; nur die zu dieser Gruppe gehörenden Objekte sind sichtbar.

Objekte gruppieren

Und wie kommt ein *View*-Objekt zu seiner Gruppe? Indem wir es explizit in die Gruppe einfügen. Dazu definieren wir zunächst die erforderlichen *Subviews*, die wir dann der Methode *Insert* übergeben. Im folgenden Beispiel kreiert *TApplication.Init* drei Objekte und verknüpft sie dann mit der Applikation:

```
InitDeskTop;
InitStatusLine;
InitMenuBar;
if DeskTop <> nil then Insert(DeskTop);
if StatusLine <> nil then Insert(StatusLine);
if MenuBar <> nil then Insert(MenuBar);
```

Einmal eingefügt werden die neu definierten Objekte Teil einer Gruppe. In diesem speziellen Fall hat *TApplication* seinen Bereich in drei voneinander getrennte Unterbereiche aufgeteilt und jedem seiner *Subviews* einen solchen Unterbereich überantwortet. Als Ergebnis erhalten Sie die schon gezeigte Aufteilung des Bildschirms: Kein Objekt überdeckt das andere.

Es gibt jedoch keinen Grund dafür, warum *View*-Objekte einander nicht teilweise oder auch ganz überdecken dürften. Im Gegenteil, gerade die Fähigkeit, einander überlappende Bildschirmfenster aufbauen und verwalten zu können, macht die Attraktivität der integrierten Entwicklungsumgebung von Turbo Pascal aus.

Eine Gruppe hält die Reihenfolge, in der *Subviews* ihr zugeordnet werden, fest. Im folgenden beziehen wir uns auf diese Reihenfolge als Z-OrdnungZ-*Ordnung*. Diese Z-Ordnung bestimmt die Reihenfolge, in der *Subviews* einerseits zur Darstellung ihrer selbst aufgefordert, und ihnen andererseits eintreffende Ereignisse zur Abarbeitung oder Weiterreichung übergeben werden.

Die Z-Ordnung näher betrachtet

Jedes *View*-Objekt wird an einer bestimmten Position innerhalb des Bereiches der Gruppe ausgegeben. Es hat außerdem eine bestimmte Ausdehnung in der X- und in der Y-Richtung. Dies ist in den Feldern *Origin* und *Size* des Objekts festgehalten. Ein solches Objekt ist damit ein zweidimensionales Gebilde. Nun können aber *View*-Objekte einander überlagern, und Turbo Vision muß wissen, welches Objekt über welchem anderen liegt. Aus diesem Grund betrachten wir eine Gruppe als ein dreidimensionales Gebilde, dessen X- und Y-Achse durch den Bereich bestimmt sind, der der Gruppe zugewiesen wurde. Die dritte Dimension ergibt sich, wenn Objekte in diese Gruppe eingefügt werden.

Die zu einer Gruppe gehörenden Subviews, »stapeln« sich entlang der Z-Achse übereinander. Das Objekt, das als erstes zur Gruppe kam, liegt

zuunterst. Das Objekt, das als letztes in die Gruppe eingefügt wurde, liegt oben auf diesem Stapel und wird auch auf dem Bildschirm über allen anderen Objekten ausgegeben. Möglicherweise überdeckt es dabei ganz oder teilweise diese anderen *View*-Objekte. Die Z-Ordnung beschreibt nichts anderes als die umgekehrte Reihenfolge, in der Objekte in eine Gruppe eingefügt wurden.

Der Bildschirm selbst ist natürlich nur eine zweidimensionale Fläche. Aber es ist ganz sinnvoll, sich bei der Darstellung einer Gruppe auf diesem zweidimensionalen Bildschirm eine dritte Dimension »dazuzudenken«. Die Abbildung 5.3 zeigt am Beispiel eines Fensters diese dritte Dimension.

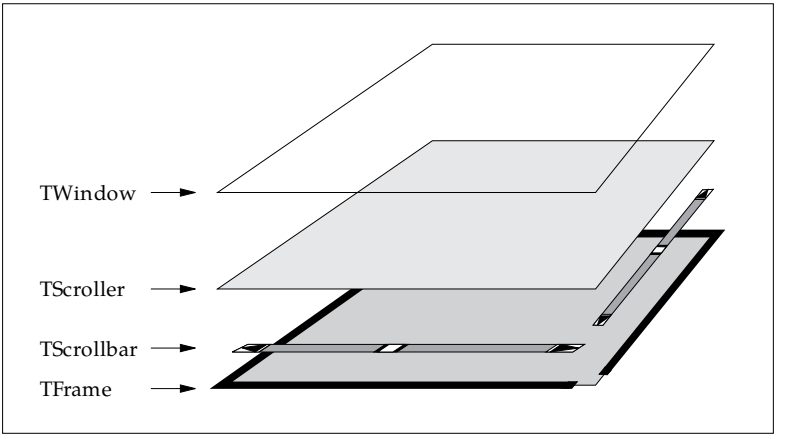


Abbildung 5.3 Das Fenster von der Seite betrachtet

Das Fenster selbst ist sozusagen eine Glasscheibe, durch die Sie eine Gruppe von *View*-Objekten sehen können. Zu dieser Gruppe kommt als erstes Objekt der Rahmen. Dann folgen Rollbalken, die Teile des Rahmens überdecken, und schließlich, als oberstes Objekt, das Fensterinnere, das den gesamten, vom Rahmen umschlossenen Bereich ausfüllt.

Auch die Arbeitsfläche, die Gruppe *TDesktop* also, ist eine solche Glasscheibe, durch die Sie die zu dieser Gruppe gehörenden Objekte sehen können.

Die erste in die Gruppe Arbeitsfläche eingefügte Subview ist das Objekt *TBackGround*. Es liegt damit zuunterst im Stapel. Dann folgen einige Fenster, die Teile dieses Objekt überdecken. Das zuletzt hinzugefügte Fenster liegt ganz oben auf dem Stapel. Aus der Perspektive der Z-Ordnung gesehen, ist dieses Fenster das erste (das oberste) Objekt, *TBackGround* das letzte (das unterste) Objekt.

Gruppenbild mit Objekten

Gruppen bilden eine Ausnahme von der allgemeinen Regel, wonach *Views* in der Lage sein müssen, sich selber darzustellen, denn gerade das tun

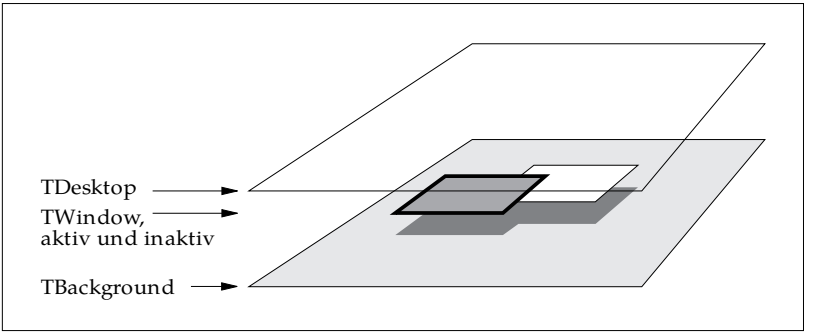


Abbildung 5.4 Ein Seitenblick auf die Arbeitsfläche

Gruppen ja nicht. Stattdessen fordern sie die zu ihnen gehörenden *View*-Objekte auf, die Ausgabe auf dem Bildschirm zu übernehmen.

Die Reihenfolge, in der diese Aufforderung an die einzelnen Objekte ergeht, ist festgelegt durch die schon besprochene Z-Ordnung. Wenn also *View*-Objekte einander überdecken, ist damit das zuletzt in die Gruppe eingefügte Objekt dasjenige, das auf dem Bildschirm als letztes ausgegeben wird und möglicherweise die anderen Objekte ganz oder teilweise überdeckt.

Da eine Gruppe selbst nichts auf dem Bildschirm ausgibt, müssen die dieser Gruppe angehörenden Objekte zusammen die gesamte Fläche überdecken, die von der Gruppe verwaltet wird. Beispielsweise bildet ein Dialogfenster eine Gruppe, zu der Rahmen, Schalter und statischer Text gehören, und diese Objekte müssen zusammen den zur Verfügung stehenden Raum ausfüllen. Andernfalls würden in dem Dialogfenster Löcher erscheinen – ein nicht gerade ästhetischer Anblick.

Ein *View*-Objekt, das Teil einer Gruppe ist, wird automatisch auf die Dimension dieses *TGroup*-Objekts zurechtgestutzt. Aber dieses mögliche Beschneiden der Bildschirmausgabe darf nicht soweit gehen, daß von dem Objekt nichts mehr zu sehen ist. Ein Teil muß immer sichtbar sein. Beispielsweise können Sie ein Fenster mit der Maus fassen und es aus der Arbeitsfläche hinausschieben, bis nur noch eine Ecke des Fensters sichtbar ist. Aber diese eine Ecke wenigstens muß noch sichtbar sein, denn sonst könnten Sie das Fenster nie mehr zurückholen. Nur der Teil einer *Subview*, der in den Begrenzungen einer Gruppe bleibt, wird auch dargestellt.

Verwandtschaftsverhältnisse und Dienstverhältnisse

Objekte sind in Turbo Vision auf zwei Arten miteinander verknüpft, und diese zwei Arten müssen Sie auf jeden Fall auseinanderhalten. Betrachten wir das einfache Dialogfenster in Abbildung 5.5. Es besteht aus einem Rahmen, einer Zeile Text und einem Schalter, der, wenn ausgelöst, das Fenster schließt. In der Terminologie von Turbo Vision handelt es sich um die Gruppe *TDialog*, zu der die Objekte *TFrame*, *TStaticText* und *TButton* gehören.

Abbildung 5.5 Ein einfaches Dialogfenster

Objekthierarchie

Auf der einen Seite sind Objekte miteinander verwandt. Sie können in dem Diagramm in Abbildung 5.6 erkennen, daß *TButton* ein direkter Nachkomme von *TView* ist, das heißt, *TButton* ist ein *TView*-Objekt, besitzt aber über die Eigenschaften, die es von *TView* geerbt hat, hinaus weitere Felder und Methoden, die seine Eigenheit ausmachen und es von *TView* unterscheiden. Auch *TDialog* ist letzten Endes ein Nachkomme von *TView*, obwohl diese Abstammungslinie über *TGroup* und *TWindow* läuft. Über diese Abstammung hat

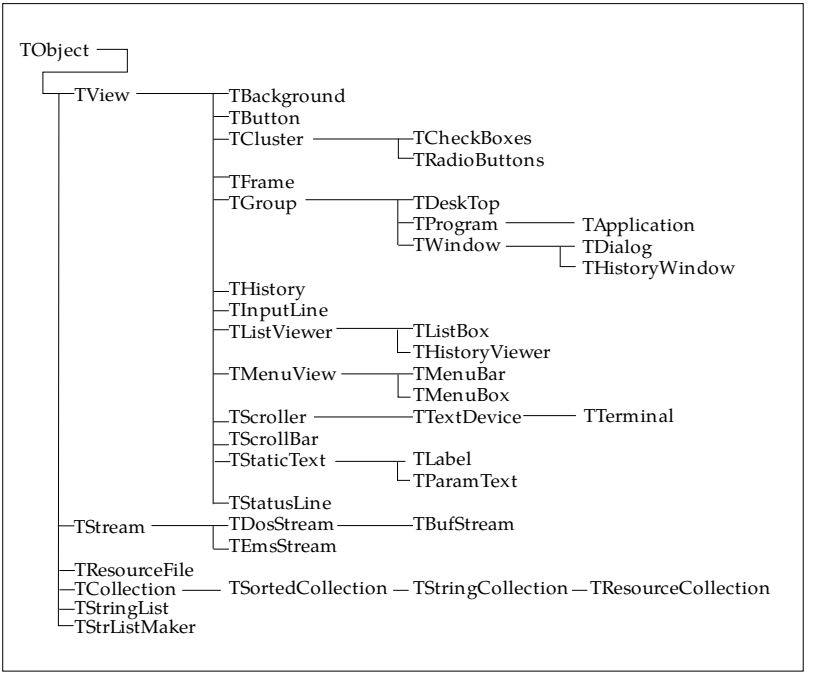


Abbildung 5.6 Objekthierarchie in Turbo Vision

TDialog vieles mit *TButton* gemein: Beide sind in der Hierarchie von Turbo Vision Verwandte, wenn auch entfernte.

Dienstverhältnisse

Objekte können auf der anderen Seite über das Verhältnis der Gruppe zu einer *Subview* miteinander in Beziehung stehen. Diese Beziehung läßt sich durchaus als Dienstverhältnis bezeichnen, denn wie erwähnt wälzt eine Gruppe die eigentliche Arbeit auf die zugehörigen *Subviews* ab. Wie im folgenden Baumdiagramm (Abbildung 5.7) gezeigt, gehört *TButton* zu *TDialog*, *TDialog* ist jedoch kein Vorfahre von *TButton*. Diese Beziehung konstituiert sich also nicht über eine gemeinsame Abstammung von Objekttypen, sondern ist eine Beziehung zwischen Instanzen von Objekten.

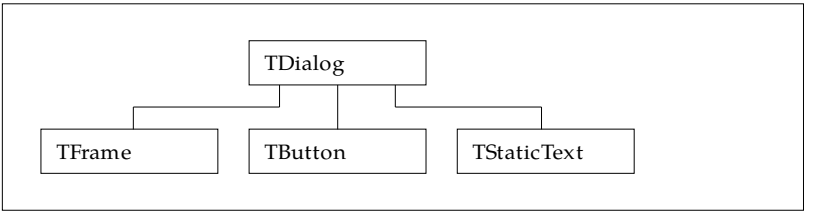


Abbildung 5.7 Baum-Diagramm eines Dialogfensters

Wenn Sie ein solches Dialogfenster programmieren, stellen *Sie* eine Verbindung zwischen *TDialog* und dem zu dieser Gruppe gehörenden Objekt *TButton* her; die Verbindung zwischen *TButton* und *TView* gründet sich dagegen auf der Objekthierarchie in Turbo Vision. Diese beiden Beziehungen dürfen Sie nicht miteinander verwechseln.

Subviews und View-Bäume

Wie schon öfter erwähnt, gehören zu *TApplication* drei weitere Objekte. Das Verhältnis dieser vier Objekte zueinander zeigt sich deutlich als baumartige Struktur, mit *TApplication* als Stamm und den Ästen Menüleiste, Arbeitsfläche und Statuszeile.

Soweit die Grundstruktur. Der Benutzer, der mit dem Programm arbeitet, wird aber zum Beispiel weitere Fenster öffnen. Diese Objekte werden mit der

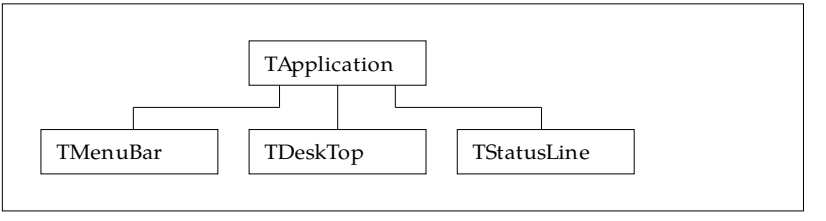


Abbildung 5.8 Grundstruktur eines Turbo-Vision-Programms

Arbeitsfläche verbunden und damit den oben dargestellten *View*-Baum um weitere Äste bereichern.

Schauen wir uns diesen Vorgang näher an. Dazu nehmen wir an, der Benutzer wähle eine Menüoption, um in einem Fenster den Inhalt einer Textdatei ansehen zu können. Dieses Fenster ist natürlich ein *View*-Objekt. Turbo Vision initialisiert dieses Objekt und ordnet es der Arbeitsfläche zu.

Zu diesem Fenster wiederum gehören andere Objekte: *TFrame*, das einen Rahmen um das Fenster zeichnet, *TScroller*, das den Text ausgibt, und ein oder zwei Rollbalken. Diese Objekte sind Teil der Gruppe Bildschirmfenster, so wie dieses Teil der Gruppe Arbeitsfläche ist.

Nun hat unser Programm also weitere *View*-Objekte definiert und erscheint wie in Abbildung 5.9 dargestellt.

Abbildung 5.9 Ein Fenster geöffnet

Aber nicht nur das Programm, auch der *View*-Baum ist komplexer geworden, wie Abbildung 5.10 zeigt.

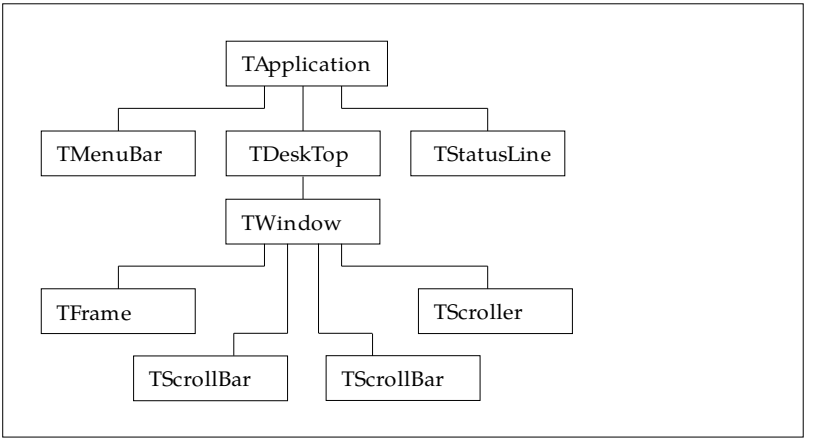


Abbildung 5.10 Ein neuer Ast: das Fenster

Der Benutzer wählt nun ein zweites Mal die gleiche Option. Ein weiteres Fenster wird erzeugt und auch der Arbeitsfläche zugeordnet. Abbildung 5.11 zeigt wiederum den Bildschirm...

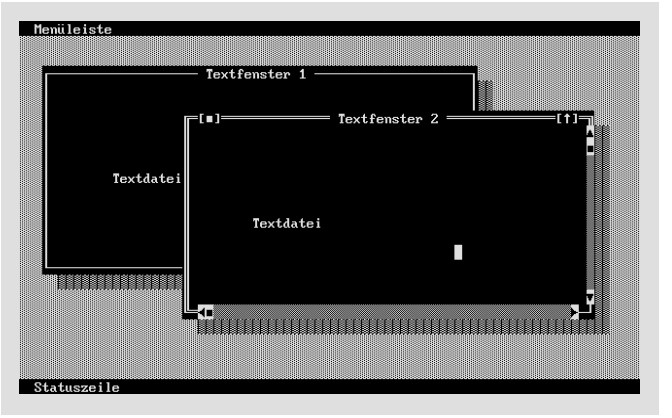


Abbildung 5.11 Ein zweites Fenster geöffnet

...und Abbildung 5.12 zeigt die neuen Verästelungen des *View*-Baums:

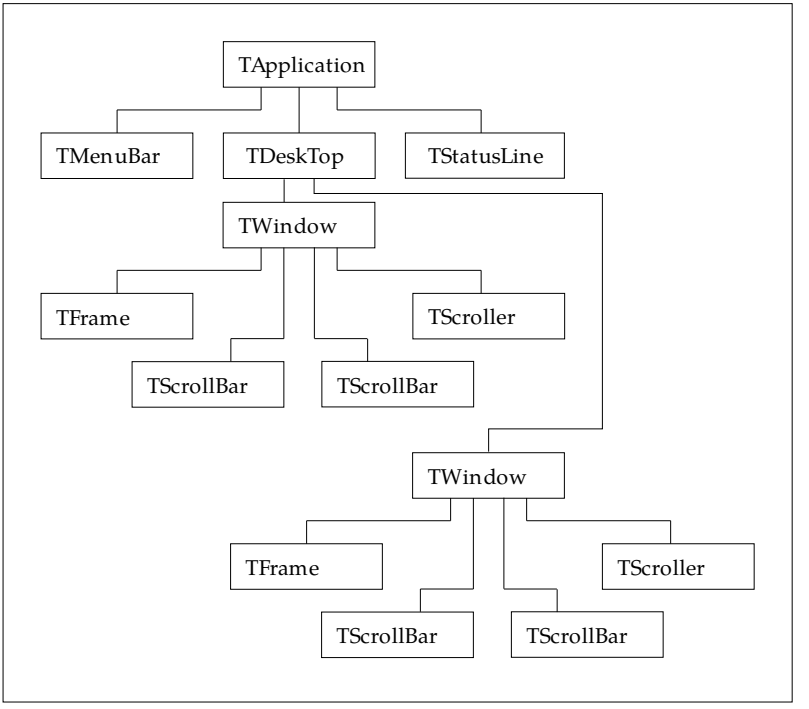


Abbildung 5.12 Die Arbeitsfläche mit zwei Fenstern

Nehmen wir nun an, der Benutzer klicke mit der Maus einen Rollbalken des aktiven Fensters an. Das *TApplication*-Objekt erkennt diesen Mausklick und erkennt weiter, daß er in einem Bereich ausgelöst wurde, der von der Arbeitsfläche kontrolliert wird. Also reicht er dieses Ereignis an die Arbeitsfläche weiter, die wiederum feststellt, daß das Fenster den betreffenden Bereich verwaltet. Aber auch damit hat das Ereignis seinen Adressaten noch nicht gefunden, sondern wird von dem Objekt Fenster an das Objekt Rollbalken weitergegeben. Erst dieses Objekt löst schließlich die dem Ereignis entsprechende Reaktion aus. Der Weg, den das Ereignis einschlagen muß, beginnt beim Stamm des *View*-Baumes, und es folgt dessen Verästelungen.

Wie dieser Vorgang genau abläuft, ist in diesem Zusammenhang unwichtig; wir gehen in Kapitel 6, »Ereignisgesteuerte Programme«, genauer darauf ein. Wichtig ist jedoch zu erkennen, wie die Objekte untereinander zusammenhängen. Und wie komplex und verästelt diese Struktur auch immer sein mag, letztlich ist jedes *View*-Objekt verbunden mit *TApplication*.

Abgebaut wird der *View*-Baum von der Spitze her. Wenn der Benutzer das Fenster schließen will, muß er entweder die Schließfeld anklicken oder die entsprechende Menüoption wählen. Zunächst veranlaßt das Fenster die Freigabe des von seinen *Subviews* belegten Speichers und gibt dann auch den von ihm selbst belegten Speicher frei. Ähnliches geschieht, wenn der Benutzer beschließt, die Arbeit mit dem Programm zu beenden. Auf den betreffenden Befehl hin, gibt *TApplication* den von seinen drei *Subviews* belegten und darauf den eigenen Speicherplatz wieder frei.

Selektierte und fokussierte Views

Jedes *View*-Objekt besitzt das Feld *State*, dessen einzelne Bits Auskunft darüber geben, welche Eigenschaften diesem Objekt zukommen. Auf zwei Bits wollen wir im folgenden unsere Aufmerksamkeit richten, die Bits *sfSelected*, das gesetzt ist, wenn das Objekt *selektiert* ist, und auf *sfFocused*, das gesetzt ist, wenn das Objekt *fokussiert* ist. Wann nun ist ein *View*-Objekt selektiert und wann fokussiert?

Nehmen wir dazu ein Beispiel: Sie öffnen ein Dialogfenster, das verschiedene Elemente enthält, eine Eingabezeile und drei Aktionsschalter. Dieses Dialogfenster ist, wie wir wissen, eine Gruppe, zu der die genannten Objekte gehören. Diese Elemente sind an sich gleichberechtigt, dennoch erkennen Sie, daß eines davon, sagen wir die Eingabezeile, farblich hervorgehoben erscheint. Sie ist das *fokussierte* Objekt und sie ist gleichzeitig das *selektierte* Objekt. Drücken Sie nun auf [Tab], und dieses Objekt verliert diese beiden Eigenschaften und auch die farbliche Hervorhebung verschwindet. Ein anderes Objekt, einer der Aktionsschalter, ist nun markiert, und dieses Objekt ist jetzt fokussiert und selektiert.

Dieser Aktionsschalter gibt, wenn betätigt, ein weiteres Dialogfenster aus. Wenn Sie nun den Schalter »drücken«, dann *verliert dieser Schalter die Eigenschaft fokussiert zu sein, bleibt aber weiterhin selektiert*. Dagegen ist nun ein

Objekt, das zu dem neu geöffneten Dialogfenster gehört, fokussiert und selektiert. Die Eigenschaft der Selektierung markiert damit Objekte auf einem Weg, an dessen Ende ein Objekt zu finden ist, das nicht nur selektiert, sondern auch fokussiert ist. Diesen Weg muß zum Beispiel ein Tastaturereignis einschlagen, wenn es das Objekt erreichen will, an das es adressiert ist. Es hangelt sich durch die als selektiert markierten Objekte, bis es ein Objekt erreicht, das auch fokussiert ist.

Erläutern wir die beiden Begriffe noch einmal an einem Beispiel, das uns allen vertraut ist: die hierarchisch organisierte Dateiverwaltung von DOS. Nehmen wir an, Sie haben auf Ihrer Festplatte C: ein Unterverzeichnis TURBO6, das neben einigen Dateien weitere Unterverzeichnisse enthält: TPGRAFIK, OOP-DEMO, PROJEKT usw. Sie wechseln aus dem Stammverzeichnis C:\ in das Unterzeichnis TURBO6, dann in das Unterverzeichnis OOPDEMO. Dort rufen Sie TURBO auf und laden die Datei BEISPIEL.PAS. In der Terminologie von Turbo Vision ist diese Datei nun fokussiert und auch selektiert. Selektiert sind aber auch die Verzeichnisse, die Sie anwählen mußten, um auf diese Datei zugreifen zu können. Den Weg dahin hält DOS im Zugriffspfad zu dieser Datei fest: C:\TURBO6\OOPDEMO\BEISPIEL.PAS.

Der Selektierungspfad

In Analogie zu diesem Zugriffspfad, nennen wir den Weg, den zum Beispiel ein Ereignis einschlagen muß, den *Selektierungspfad*. Das Ereignis wird von dem zentralen Objekt des Programms *TApplication* auf diesen Weg geschickt. *TApplication* ist eine Gruppe, weshalb zunächst das selektierte Objekt dieser Gruppe ermittelt werden muß. Ist dieses Objekt auch fokussiert, dann ist das Ziel schon erreicht. Aber dieses Objekt kann auch wieder eine Gruppe sein. Wieder muß das selektierte Objekt gefunden werden. Dies wiederholt sich so lange, bis endlich ein Objekt sowohl selektiert als auch fokussiert ist.

Abbildung 5.13 zeigt ein weiteres Beispiel für den Verlauf des Selektierungspfads. Er beginn bei dem Objekt *TDesktop*, nimmt seinen Weg über das Objekt *TWindow* und endet beim Objekt *TScroller*. Dies ist auch das fokussierte Objekt.

Wer hält den Fokus?

Das gerade fokussierte *View*-Objekt ist, wie schon erwähnt, auf dem Bildschirm in irgendeiner Weise hervorgehoben. Nehmen wir an, Sie haben mehrere Fenster geöffnet. Das gerade aktive, das fokussierte Fenster ist dasjenige, das mit einer doppelten Linie umrahmt ist; alle anderen sind nur mit einer einfachen Linie umrahmt. In einem Dialogfenster erscheint der fokussierte Schalter heller als die anderen, was bedeutet, daß dieser Schalter »gedrückt« wird, wenn Sie [Enter] betätigen.

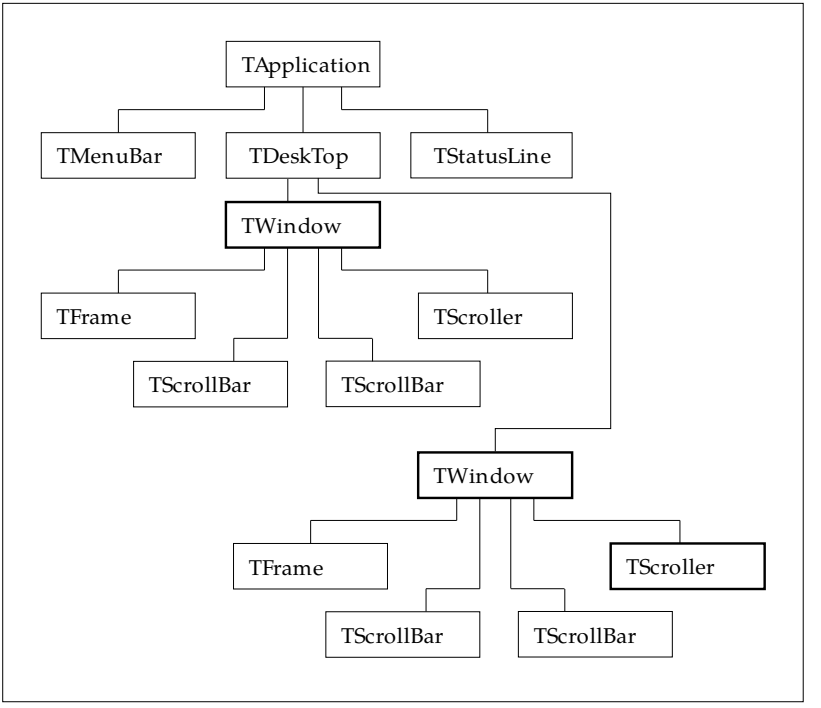


Abbildung 5.13 Der Selektierungspfad

Ein Objekt fokussieren

Wird eine Gruppe initialisiert, dann bestimmt die Gruppe, welche seiner Subviews zu fokussieren ist, indem sie die Methode *Select* dieser Subview aufruft. Damit ist das Objekt bestimmt, das gemäß Voreinstellung den Fokus hält.

Vielleicht wird der Benutzer ein anderes Objekt fokussieren wollen. Dazu kann er einfach dieses andere Objekt mit der Maus anklicken. Sind also mehrere Fenster geöffnet, dann klicken Sie das gewünschte Fenster an, woraufhin es fokussiert wird. In einem Dialogfenster bewegen Sie den Fokus mittels [Tab] von einem *View*-Objekt zu anderen. Sie können aber auch ein Objekt direkt anwählen, indem Sie es mit der Maus anklicken, oder ein Tastenkürzel benutzen, sofern ein solches Tastenkürzel für das betreffende Objekt angegeben ist. Jedoch nicht alle Objekte sind auf diese Weise ansprechbar. Der Hintergrund der Arbeitsfläche beispielsweise, der Rahmen von Fenstern und Rollbalken können nicht fokussiert werden. Bei der Initialisierung eines *View*-Objekts können Sie festlegen, ob dieses Objekt selektierbar ist oder nicht. Wenn Sie zum Beispiel den Rahmen eines Fensters anklicken, wird dieser Rahmen nicht fokussiert: dieses Objekt »weiß«, daß es den Fokus nicht übernehmen kann.

Modale View-Objekte

Ein Modus ist allgemein gesprochen die Art und Weise, in der etwas abläuft. Die integrierte Entwicklungsumgebung von Turbo Pascal kennt beispielsweise vier Modi: den Edit-Modus, den Compiler-Modus, den Run-Modus und den Debug-Modus. Der Modus, in dem sich das Programm gerade befindet, bestimmt auch die Bedeutung einzelner Tasten, sofern sie überhaupt eine Wirkung haben.

In Turbo Vision können Sie ein *View*-Objekt als *modal* erklären, und wir haben schon in Kapitel 2, »Mit Turbo Vision programmieren«, darauf hingewiesen, daß dies bei Dialogfenstern in der Regel auch notwendig ist. Wenn Sie ein solches Dialogfenster öffnen, legen Sie alles um dieses Fenster herum buchstäblich still. Der Benutzer mag die Menüleiste anwählen oder mit der Maus die Statuszeile anklicken – vergeblich. Nur Ereignisse, die das Fenster und die darin enthaltenen Objekte betreffen, werden beantwortet. Das Dialogfenster hat die Kontrolle über Ihr Programm übernommen, und diese Kontrolle gibt es erst dann wieder ab, wenn es geschlossen wird.

Ein modales *View*-Objekt beschränkt also den Gültigkeitsbereich von Eingaben des Benutzers auf sich selbst und die *Subviews*, die mit diesem Objekt verbunden sind. Solche modalen Objekte können Sie mit einer Pascal-Prozedur oder -Funktion vergleichen, in der Sie irgendwelche Variablen deklariert haben. Auch deren Gültigkeitsbereich erstreckt sich nur auf den Anweisungsblock innerhalb der Prozedur oder Funktion. Ein modales Objekt blendet alle Objekte außerhalb seiner selbst und seiner *Subviews* sozusagen aus: Ereignisse gleich welcher Art werden nicht an sie weitergegeben, sie können nicht aktiviert werden.

In einem Turbo-Vision-Programm existiert immer zumindest ein modales *View*-Objekt: es ist die Instanz von *TApplication*, das den Stamm des *View*-Baums ausmacht.

Verhaltensänderungen

Jedes *View*-Objekt besitzt vier Felder, deren einzelne Bits das Verhalten dieses Objekts bestimmen. Drei dieser Felder, *Options* (vom Typ Word), *GrowMode* und *DragMode* (beide vom Typ Byte) wollen wir im folgenden besprechen. Das vierte, *EventMask* (Typ Word), behandeln wir eingehend in Kapitel 4, »*Ereignisgesteuerte Programme*«.

Das Word Options

bestimmt über seine einzelnen Bits das allgemeine Verhalten des Objekts. In Kapitel 14, »Objekt-Referenz«, geben wir bei der Besprechung der Konstruktor-Methode eines Objekts auch an, welche Bits in *Options* bei der Initialisierung einer Instanz gesetzt werden.

Die folgende Abbildung zeigt die Zuordnung der einzelnen Bits zu den möglichen Optionen.

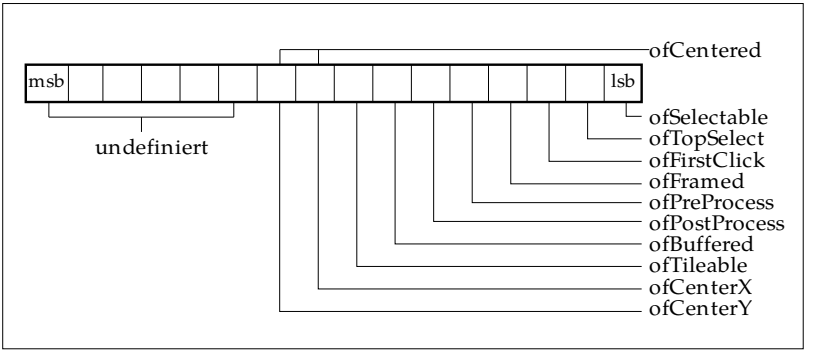


Abbildung 5.14 Die Bits in Options

Die Optionen im Detail:

ofSelectable Ist dieses Bit gesetzt, kann der Benutzer das Objekt mit der Maus selektieren. Gehört das Objekt zu einer Gruppe, ist dies zusätzlich auch mittels [Tab] möglich. Wenn Sie etwas nur zur Information auf dem Bildschirm ausgeben, soll der Benutzer ein solches Objekt nicht selektieren können. Statischer Text oder Rahmen sind in der Regel nicht selektierbar.

ofTopSelect Wenn selektiert wird das Objekt in den Vordergrund versetzt. Gedacht ist diese Option vor allem für Fenster, und sollte für Objekte, die einer Gruppe angehören, nicht genutzt werden.

ofFirstClick Der Mausklick, der dieses *View*-Objekt selektiert, wird an das Objekt selbst weitergereicht. Wird zum Beispiel ein Schalter gedrückt und dieses Bit ist gesetzt, löst ein Mausklick sowohl die Selektion des Schalters als auch die damit verbundene Aktion aus. Andere Objekte dagegen sollen nur selektiert werden können und keine weitere Aktion auslösen.

ofFramed Dieses Bit ist bei *TWindow*-Objekten gesetzt, wenn das Fenster mit einem Rahmen versehen werden soll, in allen anderen Fällen (und Objekten) ist es gelöscht.

ofPreProcess Gesetzt, wenn ein fokussiertes Ereignis zuerst an dieses

Objekt geleitet werden muß, bevor es an das fokussierte *View*-Objekt weitergereicht wird.

ofPostProcess Gesetzt, wenn fokussierte Ereignisse, die von dem fokussierten *View*-Objekt nicht abgearbeitet wurden, diesem Objekt zugeleitet werden sollen.

ofBuffered Über diese Option kann die Bildschirmausgabe einer Gruppe beschleunigt werden. Wenn eine Gruppe zum ersten Mal dargestellt wird, speichert sie automatisch auch den von ihr belegten Bildschirmbereich, sofern dieses Bit gesetzt und genügend Speicher verfügbar ist. Muß diese Gruppe ein weiteres Mal auf dem Bildschirm ausgegeben werden, kopiert die Gruppe den Puffer in den Bildschirmspeicher, anstatt die *Subviews* aufzufordern, sich selbst auszugeben. Wenn ein Aufruf von *New* oder *GetMem* nicht mehr genügend Speicher vorfindet, beginnt die Speicherverwaltung automatisch, die belegten Puffer wieder freizugeben, bis die Speicheranforderung erfüllt werden kann.

Verfügt eine Gruppe über einen Puffer, dann stoppt ein Aufruf der Methode *Lock* die Übertragung des Pufferinhalts in den Bildschirmspeicher solange, bis *Unlock* aufgerufen wird. Dies vermeidet ein Flackern des Bildschirms während komplexer Ausgaben auf dem Bildschirm. Die Arbeitsfläche beispielsweise geht so vor, wenn sie ihre *Subviews* geordnet nebeneinander oder kaskadiert darstellen muß (siehe dazu auch die folgende Option).

ofTileable Die Arbeitsfläche kann die gerade geöffneten Fenster geordnet nebeneinander oder fächerartig hintereinander darstellen. Wollen Sie, daß ein Fenster von diesen Möglichkeiten ausgeschlossen ist, dann löschen Sie dieses Bit. Dieses Fenster wird dann dort verharren, wo es sich gerade befindet, während die anderen Objekte in der genannten Weise angeordnet werden.

Sie können die *View*-Objekte einer Gruppe sehr einfach über *TApplication.HandleEvent* nebeneinander oder kaskadiert darstellen lassen:

```
cmTile:
  begin
    DeskTop^.GetExtent(R);
    DeskTop^.Tile(R);
  end;
cmCascade:
  begin
    DeskTop^.GetExtent(R);
    DeskTop^.Cascade(R);
  end;
```

Sind mehr Objekte vorhanden, als kaskadiert dargestellt werden können, wird der Befehl von der Arbeitsfläche ignoriert.

ofCenterX Wenn in die Gruppe eingefügt, wird das *View*-Objekt in X-Richtung zentriert.

ofCenterY Wenn in die Gruppe eingefügt, wird das *View*-Objekt in Y-Richtung zentriert. Diese Option ist dann besonders wichtig, wenn ein Bildschirmfenster sowohl im 25- als auch im 43-Zeilen-Modus des Bildschirms korrekt dargestellt werden soll.

ofCentered Wenn in die Gruppe eingefügt, wird das *View*-Objekt sowohl in X- als auch in Y-Richtung zentriert.

Das Byte GrowMode

legt fest, wie ein *View*-Objekt reagiert, wenn die Größe der Gruppe geändert wird. Die einzelnen Bits haben folgende Bedeutung:

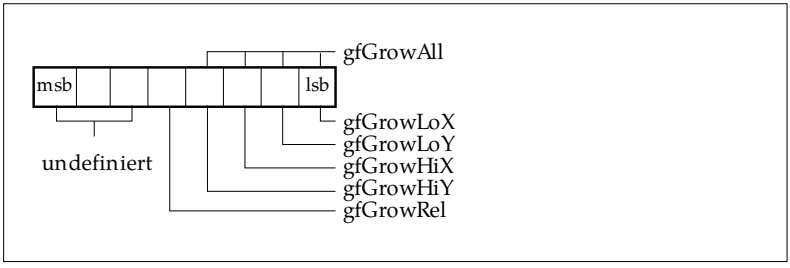


Abbildung 5.15 Die Bits in GrowMode

gfGrowLoX Wenn dieses Bit gesetzt ist, bewahrt das *View*-Objekt eine konstante Distanz von der linken Seite der Gruppe.

gfGrowLoY Wenn dieses Bit gesetzt ist, bewahrt das *View*-Objekt eine konstante Distanz von der oberen Seite der Gruppe.

gfGrowHiX Wenn dieses Bit gesetzt ist, bewahrt das *View*-Objekt eine konstante Distanz von der rechten Seite der Gruppe.

gfGrowHiY Wenn dieses Bit gesetzt ist, bewahrt das *View*-Objekt eine konstante Distanz von der unteren Seite der Gruppe.

gfGrowAll Wenn dieses Bit gesetzt ist, ändert das *View*-Objekt seine Größe nicht und bewegt sich zusammen mit der unteren rechten Ecke der Gruppe.

gfGrowRel Das *View*-Objekt ändert seine Größe im gleichen Verhältnis, in dem auch die Gruppe ihre Größe ändert. Diese Option

sollten Sie nur bei *TWindow* und dessen Nachkommen verwenden, wenn solche Objekte zur Gruppe Arbeitsfläche gehören. Das Fenster bewahrt dann seine relative Größe, wenn zwischen 25- und 43/50-Zeilen-Modus umgeschaltet wird. Verwenden Sie diese Option nicht für *View*-Objekte eines Fensters.

Das Byte DragMode

legt fest, ob das Objekt seine Größe ändern und über den Bildschirm bewegt werden kann. Bei der Initialisierung eines *View*-Objekts wird das Bit *dmLimitLoY* gesetzt; der obere Teil des Objekts kann damit den Bereich der Gruppe nicht verlassen.

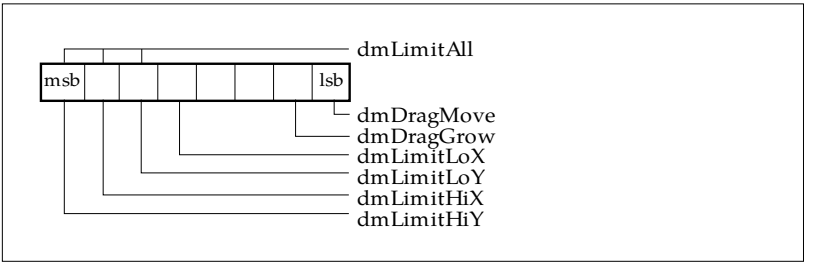


Abbildung 5.16 Die Bits im DragMode

Die einzelnen Bits haben folgende Bedeutung:

dmDragMove Ist dieses Bit gesetzt, können Sie das *View*-Objekt mit der Maus bewegen, wenn Sie den oberen Rahmen angeklickt haben.

dmDragGrow Ist dieses Bit gesetzt ist, kann das *View*-Objekt seine Größe ändern.

dmLimitLoX Ist dieses Bit gesetzt, kann die linke Seite des *View*-Objekts den Bereich der Gruppe nicht verlassen.

dmLimitLoY Ist dieses Bit gesetzt, kann die obere Seite des *View*-Objekts den Bereich der Gruppe nicht verlassen.

dmLimitHiX Ist dieses Bit gesetzt, kann die rechte Seite des *View*-Objekts den Bereich der Gruppe nicht verlassen.

dmLimitHiY Ist dieses Bit gesetzt, kann die untere Seite des *View*-Objekts den Bereich der Gruppe nicht verlassen.

dmLimitAll Ist dieses Bit gesetzt, kann das *View*-Objekt den Bereich der Gruppe nicht verlassen.

Der Status eines View-Objekts

wird in *State* festgehalten, dessen einzelne Bits Auskunft darüber geben, ob dieses Objekt sichtbar ist, ob es gerade über den Bildschirm bewegt wird usw. Die Bedeutung der einzelnen Bits in *State* beschreiben wir im Eintrag »sfXXXX-Konstanten« in Kapitel 15, »Außerhalb der Objekte«.

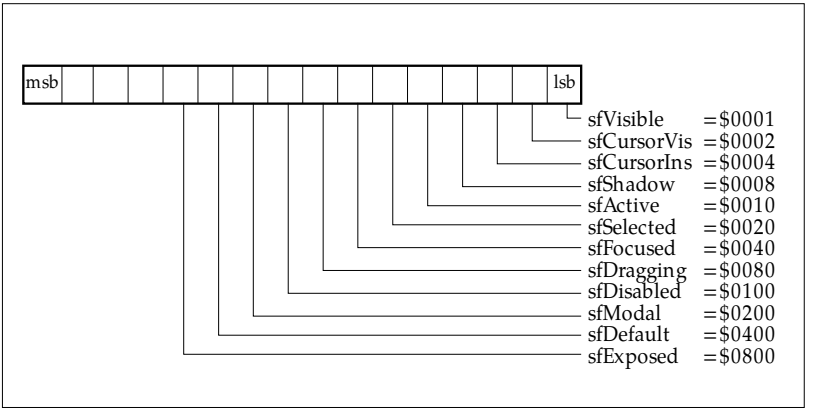


Abbildung 5.17 Die Bits in State

Über die in jedem *View*-Objekt definierte Methode *SetState* ändert Turbo Vision selbst dieses Status-Flag immer dann, wenn dieses Objekt fokussiert wird, den Fokus abgibt oder selektiert wird.

Der Aufruf der Methode *SetState* ist oft mit einer Aktion verbunden. Ein Schalter beispielsweise ändert seine Farbe in türkis, sobald er fokussiert wird. Der folgende Programmauschnitt zeigt eine *SetState*-Methode für einen Nachkommen von *TView*:

```
procedure TButton.SetState (AState : Word; Enable : Boolean);
begin
  TView.SetState(AState, Enable);
  if AState = sfSelected then DrawView;
end;
```

Wie Sie an diesem Beispiel sehen können, sollten Sie innerhalb von *SetState* die *SetState*-Methode des Vorfahren aufrufen.

Wenn das Verhalten Ihres Programms sich abhängig vom Status eines *View*-Objekts ändern soll, dann müssen Sie die Methode *SetState* dieses Objekts überschreiben. Angenommen, Ihr Programm stelle einen Texteditor bereit, und Sie wollen – abhängig davon, ob ein Editorfenster geöffnet ist oder nicht – alle Menüoptionen, die sich auf diesen Editor beziehen, aktivieren oder deaktivieren. Überschreiben Sie dann *SetState* wie im folgenden Beispiel gezeigt:

```
procedure TEditor.SetState(AState : Word; Enable Boolean);
const
  EditorCommands = (cmSearch, cmReplace, cmSearchAgain, cmGotoLine,
                    cmFindProc, cmFindError, cmSave, cmSaveAs);
begin
  TView.SetState(AState, Enable);
  if AState and sfActive <> 0 then
    if Enable then EnableCommands(EditorCoammnds)
    else DisableCommands(EditorCommands);
end;
```

Nicht immer kann Turbo Vision alles von sich aus erledigen. Dann sind Sie als Programmierer gefordert. Nehmen wir an, Sie wollen, daß ein Blockcursor erscheint, sobald der Benutzer den Editor vom Einfüge- in den Überschreibmodus umschaltet.

Dazu muß zunächst natürlich eine Taste definiert werden, über die dieser Modus zu ändern ist; verwenden wir in unserem Fall die Taste [Ins]. Drückt der Benutzer diese Taste und das Editorfenster ist die fokussierte *View*, wird das Ereignis »[Ins] gedrückt« an den Texteditor übermittelt. Dessen *HandleEvent*-Methode reagiert auf das Ereignis, indem sie intern festhält, daß auf den Überschreibmodus umgeschaltet wurde, und ruft die Methode *BlockCursor* auf. Diese Methode wiederum benutzt die *SetState*-Methode der Editor-*View*, um das Flag *sfCursorIns* auf False zu setzen. Ab jetzt kann wieder Turbo Vision übernehmen: Da der Texteditor ein Nachkomme von *TView* ist, reagiert die *SetState*-Methode auf diese Statusänderung, indem sie die Änderung der Form des Cursors veranlaßt.

In welchen Farben ausgeben

Wir wissen: Über Geschmack ist nicht zu streiten. Und kaum zwei Leute sind sich einig darüber, welche Bildschirmfarben zu verwenden seien. Turbo Vision erlaubt Ihnen, die Ausgabefarben zu verwenden, die Ihren Vorstellungen entsprechen. Realisiert wird diese Möglichkeit über das Konzept der *Farbpaletten*.

Farbpaletten

Wird ein *View*-Objekt aufgefordert, sich auf dem Bildschirm darzustellen, erwartet es auch eine Information über die zu verwendende Farbe, jedoch nicht als direkten Farbwert, sondern als Index einer Farbtabelle. Diese Farbtabelle sieht zum Beispiel für das Objekt *TScroller* folgendermaßen aus:

```
CSroller  = #6#7;
```

Wie Sie sehen, sind Farbpaletten als Konstanten des Typs String definiert, denn dieser Datentyp erlaubt uns, Paletten unterschiedlicher Länge anzulegen. Im gegebenen Fall ist *CScroller* ein String von zwei Zeichen Länge, und diese Zeichen sind nichts anderes als die Einträge in der Farbpalette. Die Zuordnung dieser Einträge sieht folgendermaßen aus:

```
{ Farbzuordnung     }
{ 1 = Normal        }
{ 2 = Hervorgehoben }
```

Etwas deutlicher geht dieser Zusammenhang aus der folgenden Abbildung hervor:

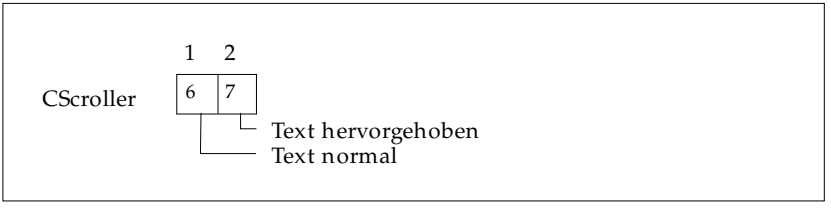


Abbildung 5.18 Farbpalette von TScroller

Einem Objekt *TScroller* stehen somit zwei Farben zur Verfügung, eine Normalfarbe und eine Farbe für Hervorhebungen, die jeweils durch die Einträge in der Palette festgelegt sind. Wird normaler Text ausgegeben, ruft die *Draw*-Methode des Objekts die Methode *GetColor* mit dem Parameter 1 auf, wodurch über den ersten Eintrag in der Palette die Ausgabefarbe ermittelt wird. Ein Aufruf von *GetColor*(2) gibt die Farbe für hervorgehobene Teile zurück.

Wenn Sie mit den voreingestellten Bildschirmfarben zufrieden sind, ist dies alles, was Sie über Farbpaletten wissen müssen. Die in Turbo Vision voreingestellten Farben wurden so gewählt, daß bei einer Kombination verschiedener *View*-Objekte auch die verwendeten Farben möglichst zueinander passen.

Farbpaletten – genauer betrachtet

Welche Funktion haben nun die Einträge in den Paletten selbst? Auch ein solcher Eintrag gibt in der Regel nicht den Farbwert selbst wieder, sondern ist als Index in die Farbpalette der Gruppe aufzufassen. Der erste Eintrag in der Farbpalette eines *TScroller*-Objekts hat den Wert 7. Nehmen wir nun an, dieses Objekt gehöre zu einer Gruppe *TWindow*. Wie gesagt – dieser Wert ist kein Farbwert, sondern eine Referenz auf den siebten Eintrag der Farbpalette des Fensterobjekts.

Wir sehen, daß der siebte Eintrag in der Palette von *TWindow* den Wert $1E enthält. Dieser Wert ist wiederum ein Index in die Palette der Gruppe, der das Fenster angehört, und diese Gruppe ist das Programm selbst. Damit

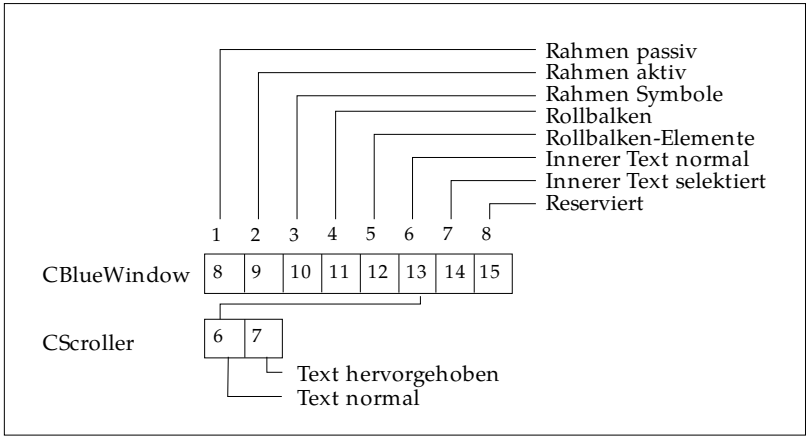


Abbildung 5.19 Zuordnung der Farbpalette von TScroller zu der von TWindow

sind wir am Ende des Weges angelangt und können den Wert $1E als den zu verwendenden Farbwert betrachten. Dieses Farbattribut entspricht der Hintergrundfarbe 1 und der Vordergrundfarbe $E (dezimal 14): Die Zeichen werden in hellem Gelb auf Blau dargestellt. Denken Sie aber besser nicht an solche Farbkombinationen, denn es ist sehr viel eingängiger, wenn Sie sagen, Ihr Text solle in der Farbe ausgegeben werden, die für Text in Fenstern definiert ist. Denken Sie also nicht an die Farben selbst, sondern an die Objekte, die diese Farben bei ihrer Darstellung benutzen.

Die Methode GetColor

Die Ermittlung des Farbwertes ist Aufgabe der in *TView* definierten virtuellen Methode *GetColor*. Diese Methode muß sich, ausgehend von dem Objekt, das darzustellen ist, den *View*-Baum von Gruppe zu Gruppe hinunterhangeln, bis es das *TApplication*-Objekt erreicht. Bei jedem Objekt, das ihr dabei in den Weg kommt, ruft sie die zuständige Methode *GetPalette* auf. Am Ende des Weges steht schließlich ein Bildschirmattribut.

Die Farben ändern

Wollen Sie die verwendeten Farben ändern, müssen Sie die Einträge in den Paletten ändern. Das ist soweit klar. Wenn also die Normalfarbe für Rollbalken beispielsweise Ihrem Geschmack nicht entspricht, werden Sie zunächst den ersten Eintrag in der Farbpalette dieses Objekts ändern. Statt des voreingestellten Wertes 6 nehmen Sie den Wert 5. Aber: Sie haben damit nur die Zuordnung der Normalfarbe des Rollbalkens zu den Farben, die das Fenster

verwendet, geändert. Der Rollbalken erscheint jetzt in der Farbe, die das Fenster für die Darstellung der Pfeile am Rande des Rollbalkens verwendet, und das sind in der Voreinstellung blaue Zeichen auf türkisfarbenem Hintergrund.

Wie gehen Sie aber vor, wenn Sie die Farbkombination helles Gelb auf Blau ändern wollen? Sie ändern den Eintrag für Normaltext in der Palette von *TWindow*. Da die Palette der Gruppe *TWindow* die letzte ist, die definierte Einträge enthält, legen diese Einträge die tatsächlichen Farben fest, die zur Darstellung aller zu einem Fenster gehörenden Objekte verwendet werden. Denken Sie immer daran: Paletten-Einträge sind in der Regel keine direkten Farbattribute; diese werden vielmehr bestimmt durch die Farbpalette der Gruppe, der ein Objekt angehört.

Wozu eigentlich diese Hangelei durch Farbpaletten? Nun, Sie wollen wahrscheinlich, daß die Fenster in Ihrem Programm ähnlich aussehen, und Sie wollen sich wohl kaum die Mühe machen, bei jedem einzelnen Fenster die verwendete Farbe anzugeben. Unterstellen wir aber, Sie machen sich diese Mühe. Irgendwann ändern Sie schließlich Ihre Meinung, oder Sie ermöglichen es den Benutzern Ihrer Programme gar, die Farben selbst festzulegen. Spätestens dann würde sich erweisen, wie unflexibel ein solches Vorgehen ist. Das ist aber nur die eine Seite. Ein anderer Vorteil der in Turbo Vision verwendeten Methode besteht darin, daß Sie sich nicht darum kümmern müssen, in welcher Farbe ein *View*-Objekt innerhalb einer Gruppe erscheinen wird. Sie können das gleiche Objekt einem Fenster, später einem Dialog zuordnen. Es wird in jeweils anderen Farben dargestellt, aber immer ist gewährleistet (jedenfalls sofern Sie die voreingestellten Farben verwenden), daß die Farben des Objekts mit den Farben der Gruppe harmonieren.

Sie ändern die Palette eines *View*-Objekts, indem Sie die Methode *GetPalette* dieses Objekts überschreiben. Nehmen wir an, Sie definieren ein neues Objekt, das Sie von *TScroller* ableiten. Bei der Ausgabe der Rollbalken soll dieses Objekt die Farbe verwenden, die ein Fenster-Objekt für den Rahmen gebraucht. Der folgende Programmausschnitt zeigt Ihnen, wie Sie dann vorgehen müssen.

```
type
  TMyScroller = object(TScroller)
    function GetPalette: PPalette; virtual;
  end;

function TMyScroller.GetPalette: PPalette;
const
  CMyScroller = #1#7;
  PMyScroller: String[Length(CMyScroller)] = CMyScroller;
begin
  GetPalette := @PMyScroller;
end;
```

Die Farbpalette ist als Konstante des Datentyps String definiert. Diese Definition erlaubt, die Paletten in sehr effizienter Weise zu bearbeiten, da alle Stringfunktionen auch auf Paletten angewendet werden können.

Neue Farben hinzufügen

Vielleicht wollen Sie den schon zur Verfügung stehenden Farben weitere hinzufügen, um die Möglichkeiten der Darstellung zu erweitern. Zum Beispiel sollen die Rollbalken über eine weitere Hervorhebungsfarbe verfügen, wie das schon das Breakpoint-Fenster innerhalb der integrierten Entwicklungsumgebung realisiert. Dazu müssen Sie zunächst einen neuen Objekttyp von *TWindow* ableiten und die geerbte Farbpalette erweitern:

```
type
  TMyWindow = object(TWindow)
    function GetPalette: PPalette; virtual;
  end;

function TMyWindow.GetPalette: PPalette;
const
  CMyWindow = CWindow + #$4F;
  P: string[Length(CMyWindow)] = CMyWindow;
begin
  GetPalette := @P;
end;
```

CWindow ist eine Stringkonstante, die die voreingestellte Palette von *TWindow* speichert. Durch Ihre Erweiterung verfügt *TMyWindow* über einen neuen Eintrag in der Palette, den Sie als Hervorhebungsfarbe nutzen können. Zuvor aber müssen Sie auch die Methode *TMyScroller.GetPalette* ändern:

```
function TMyScroller.GetPalette: PPalette;
const
  CMyScroller = #1#8#9;
  P: String[Length(CMyScroller)] = CMyScroller;
begin
  GetPalette := @P;
end;
```

Über den dritten Eintrag in der Palette können Sie jetzt auf die neue Farbe – leuchtendes Weiß auf Rot – zugreifen. Vergewissern Sie sich aber, daß die Gruppe, zu der *TMyScroller* gehört, tatsächlich die erweiterte Palette *CMyWindow* verwendet. Sollten es Ihnen doch passieren, daß Sie auf das neunte Element einer Palette zugreifen, die tatsächlich nur acht Elemente umfaßt, dann erscheint die Ausgabe in leuchtendem, blinkenden Weiß vor rotem Hintergrund. Das ist der Farbwert, der in der Variablen *ErrorAttr* (vergleichen Sie Kapitel 15, »Außerhalb der Objekte«) gespeichert ist. Daran können Sie erkennen, daß irgend etwas bei der Zuordnung der Farben schiefgelaufen ist.

6. Ereignisgesteuerte Programme

Wir haben Turbo Vision in der Absicht entwickelt, Ihnen ein Gerüst für die Entwicklung fensterorientierter und ereignisgesteuerter Programme zur Verfügung zu stellen. In den vorigen Kapiteln konnten Sie feststellen, daß die Charakterisierung »fensterorientiert« gegenüber dem, was Turbo Vision tatsächlich bietet, eine ziemliche Untertreibung darstellt. In diesem Kapitel betrachten wir die zweite Eigenschaft, die ein Turbo-Vision-Programm auszeichnet, genauer. Was heißt das eigentlich, ein Programm wird durch Ereignisse gesteuert?

Turbo Vision mit Leben erfüllen

Wenn Turbo Vision ein Fenster öffnet, die Menüleiste aktiviert oder ein Untermenü ausgibt, dann steckt immer ein Ereignis dahinter. Ereignisse steuern die Aktivitäten eines Turbo-Vision-Programms, denn es reagiert auf Ereignisse. Solche Ereignisse können auf Eingaben des Benutzers zurückgehen, sie können aber auch durch andere Komponenten Ihres Programms ausgelöst worden sein.

Eingaben lesen

In einem traditionellen Pascal-Programm programmieren Sie eine Schleife, innerhalb derer Sie Eingaben von der Tastatur, der Maus oder einem anderen Eingabegerät entgegennehmen. Ebenfalls in dieser Schleife wird die Eingabe bewertet, dann die passenden Prozeduren oder Funktionen aufgerufen. Oder Ihr Programm verzweigt in eine andere Schleife, die ihrerseits wieder Eingaben entgegennimmt:

```
repeat
  B := ReadKey;
  case B of
    'i' : InvertArray;
```

```
      'e' : EditArrayParams;
      'g' : GraphikDiaplay;
      'q' : Quit := true;
    end;
  until Quit;
```

Auch ein ereignisgesteuertes Programm ist nicht wesentlich anders strukturiert. Und ein Programm, dessen Ablauf durch Eingaben des Benutzers gesteuert wird, muß immer irgendwie diese Eingaben entgegennehmen, sie bewerten, darauf reagieren. Ein ereignisgesteuertes Programm jedoch tritt Ihnen, dem Programmierer, anders gegenüber.

In einem Turbo-Vision-Programm müssen Sie sich nicht mehr darum kümmern, Eingaben des Benutzers entgegenzunehmen. Das erledigt Turbo Vision für Sie. Eine solche Eingabe wird abgespeichert in einem Record des Typs TEvent, und dieser Record wird dann an Ereignis-Recorddas zuständige *View*-Objekt weitergereicht. Die Programmteile, die Sie schreiben, müssen also mit den sie betreffenden Eingaben umgehen und darauf reagieren können.

Nehmen wir zum Beispiel an, der Benutzer klicke ein gerade inaktives Fenster an. Turbo Vision registriert dieses Mausereignis, speichert es in einem *TEvent*-Record und übergibt diesen Record dem inaktiven Fenster.

»Schön,« werden Sie jetzt vielleicht denken, »um die Eingaben des Benutzers brauche ich mich also nicht zu kümmern. Was ich dagegen lernen muß, ist, einen solchen *TEvent*-Record auszuwerten und dann zu veranlassen, daß das Fenster aktiviert wird.« Aber auch dabei nimmt Ihnen Turbo Vision viel Arbeit und in diesem speziellen Fall sogar die ganze Arbeit ab.

View-Objekte können auf Eingaben des Benutzers selbständig reagieren. Ein Fenster »weiß«, wie es sich öffnet, seine Größe oder Position verändert, sich wieder schließt. Die Menüleiste kann in Reaktion auf Eingaben des Benutzers Untermenüs öffnen und wieder schließen oder eine andere verfügbare Option markieren. Sie können es also dem Fenster selbst überlassen, sich zu aktivieren. Aber wie geben sich Ereignisse nun Ihrem Programm zu erkennen, und was erkennt Turbo Vision überhaupt als Ereignis an?

Das Wesen von Ereignissen

In der Repeat-Schleife im vorigen Abschnitt wurde nur eine Art von Ereignissen entgegengenommen: ein Tastendruck. Turbo Vision ist wesentlich flexibler. Es unterstützt neben der Tastatur auch die Maus als Eingabegerät. Neben diesen zwei externen Quellen von Ereignissen gibt es noch eine dritte Quelle, das Programm selbst. Jede Aktion der Maus, jeder Tastendruck, jede Botschaft aus einer Komponente des Programms stellt für Turbo Vision ein Ereignis dar. Und dieses Ereignis ist eine nicht weiter aufteilbare Einheit. Wenn also der Benutzer ein Wort von sagen wir fünf Zeichen Länge eintippt, sind das fünf separate Ereignisse.

Jedes dieser Ereignisse speichert Turbo Vision in einem Record des Typs *TEvent*, dessen Feld *What* über die Art des Ereignisses Auskunft gibt. Die anderen Felder spezifizieren das Ereignis näher. Sie enthalten beipielsweise den Tastencode, wenn ein Tastendruck vorliegt. Oder sie halten die Koordinaten der Position des Mauszeigers fest, wenn es sich um ein Mausereignis handelt.

Da nun je nach Ereigniskategorie ein solcher *TEvent*-Record anders verarbeitet wird, müssen wir zunächst einmal genauer betrachten, welche Arten von Ereignissen Turbo Vision überhaupt definiert.

Kategorien von Ereignissen

Dazu betrachten wir das numerische Feld *What* eines *TEvent*-Records etwas genauer. Dieses Feld ist vom Typ Word, und die einzelnen Bits dieses Word dienen dazu, die verschiedenen Arten von Ereignissen zu spezifizieren. Die folgende Abbildung zeigt die Bitpositionen, die Turbo Vision schon benutzt. Die Bezeichnungen an der rechten Seite der Grafik sind die Namen vordefinierter Konstanten, die unter anderem dazu dienen, in *TEvent.What* ein bestimmtes Bit zu setzen.

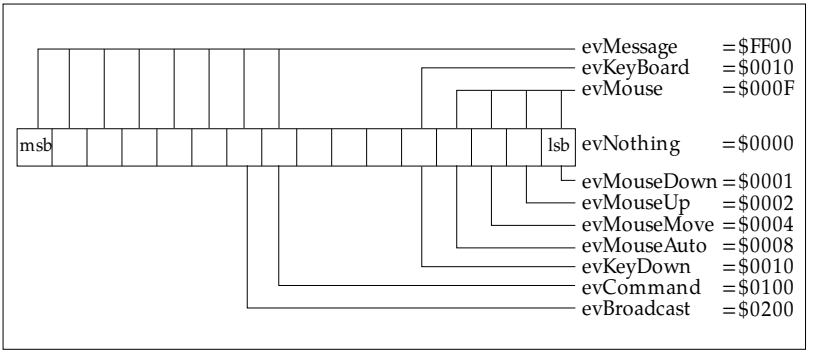


Abbildung 6.1 Die Bits von TEvent.What

Die nach oben geführten Verzweigungen in der Grafik richten sich auf Konstantennamen, die für die drei Kategorien von Ereignissen stehen, die Turbo Vision erkennt. *evMouse* bezeichnet Maus-, *evKeyBoard* Tastaturereignisse und *evMessage* Botschaften. Die Konstante *evNothing* schließlich dient dazu, schon bearbeitete Ereignisse zu bezeichnen. In diesem Fall ist in *TEvent.What* kein Bit gesetzt.

Mausereignisse

Die Kategorie Mausereignisse umfaßt, so wie in Turbo Vision definiert, vier Arten von Ereignissen, die mittels der Maus ausgelöst werden. Drückt der Benutzer die linke Maustaste, wird ein *evMouseDown*-, läßt er die Taste los, ein

evMouseUp-Ereignis erzeugt. Ein Ereignis des Typs *evMouseMove* zeigt, daß die Maus bewegt wurde. Hält der Benutzer die Maustaste gedrückt, generiert Turbo Vision in kurzen Abständen Ereignisse des Typs *evMouseAuto*. Dadurch kann zum Beispiel ein Objekt, das mittels der Maus über den Bildschirm bewegt wird, Positionsänderungen verfolgen und, wenn notwendig, sich neu auf dem Bildschirm darstellen. Jedes Mausereignis übermittelt die Position des Mauszeigers im *TEvent*-Record, so daß das Objekt, das auf ein solches Ereignis reagieren muß, erfahren kann, wo dieses ausgelöst wurde.

Die Konstante *evMouse*, die alle vier besprochenen Ereignisse zusammenfaßt, können Sie benutzen, um zu überprüfen, ob gerade ein Mausereignis vorliegt. Statt also wie in der folgenden Anweisung über die oben genannten Bitmasken zu ermitteln, ob überhaupt ein Mausereignis vorliegt:

```
if Event.What and (evMouseDown or evMouseUp or evMouseMove
  or evMouseAuto) <> 0 then ...
```

schreiben Sie einfach

```
if Event.What and evMouse <> 0 then ...
```

Tastaturereignisse

sind verglichen mit Mausereignissen wesentlich einfacher. Sie drücken eine Taste, und Turbo Vision trägt in das *What*-Feld des *TEvent*-Records den Wert *evKeyDown* ein und speichert in einem anderen Feld den Code der betreffenden Taste ab.

Botschaften

In dieser Kategorie definiert Turbo Vision selbst bereits zwei Arten: Befehlsereignisse des Typs *evCommand* und Rundrufereignisse des Typs *evBroadcast*. Sie unterschieden sich darin, wie Turbo Vision sie behandelt. Wir werden später näher darauf eingehen. Die obige Abbildung zeigt, daß in dieser Kategorie noch sechs Bitpositionen frei sind, die Sie zur Definition spezifischer Ereignisarten benutzen können.

Erledigte Ereignisse

Bleibt noch die besondere Kategorie von Ereignissen zu besprechen, die keine mehr sind. Speichert das *What*-Feld eines *TEvent*-Records den Wert *evNothing*, bedeutet dies, daß dieses Ereignis schon abgearbeitet wurde und keiner weiteren Beachtung mehr bedarf.

Hat ein Objekt in Turbo Vision ein Ereignis bearbeitet, ruft es die globale Prozedur *ClearEvent* auf, die das Feld *What* auf den Wert *evNothing* setzt. Solche Ereignisse sollten von anderen Objekten nicht mehr weiter beachtet werden. Aber *ClearEvent* macht noch mehr. Diese Routine setzt das Feld *TEvent.InfoPtr* auf den Wert *@Self*, wodurch sich das Objekt, das dieses Ereignis bearbeitet hat, zu erkennen gibt. Wir werden später sehen, daß diese Information in bestimmten Fällen ganz nützlich ist.

Ereignisse und Befehle

Die meisten Ereignisse werden schließlich in ein Ereignis des Typs *evCommand* verwandelt. Beispielsweise führt ein Mausklick auf eine Option in der Statuszeile dazu, daß ein *TEvent*-Record auf den Weg geschickt wird, dessen *What*-Feld Auskunft über das Ereignis gibt. Schließlich empfängt das *TStatusLine*-Objekt diesen Record und antwortet damit, seinerseits einen solchen Record zu erzeugen, dessen *What*-Feld auf den Wert *evCommand* gesetzt ist, und der darüber hinaus auch den Befehl speichert, der mit der betreffenden Option verbunden wurde.

Ereignisse und ihr Weg

Woher sie kommen...

Wie schon öfter erwähnt, ist ein Turbo-Vision-Programm eine Instanz des Objekts *TApplication*, und dessen *Run*-Methode ruft *TGroup.Execute* auf, die im Kern eine Repeat-Schleife darstellt, wie sie der folgende Programmauschnitt zeigt:

```
var E: TEvent;

E.What := evNothing;         { kein Ereignis aufgetreten }
repeat
  if E.What <> evNothing then EventError(E);
  GetEvent(E);                    { ein Ereignis holen... }
  HandleEvent(E);  { ...und zur Bearbeitung weitersenden }
until EndState <> Continue;
```

Die Methode *GetEvent* überprüft, ob ein Ereignis vorliegt und belegt – sollte dies der Fall sein – den *TEvent*-Record mit den passenden Werten. Diesen Record sendet *HandleEvent* an die zuständigen *View*-Objekte. Diese müssen auf das Ereignis reagieren und dessen erfolgreiche Bearbeitung dadurch dokumentieren, daß sie das Feld *What* des Records auf den Wert *evNothing* setzen. Es kann aber auch sein, daß ein Ereignis keinen zuständigen Sachbearbeiter findet. Dann wird, sobald das nicht bearbeitete Ereignis wieder zurückkehrt, *EventError* aufgerufen. Sofern nicht überschrieben, beschränkt sich diese Methode darauf, gar nichts zu tun.

Eine detailliertere Besprechung der Methoden *GetEvent*, *HandleEvent* und *EventError* heben wir uns für später auf.

...wohin sie gehen

Der Weg aller Ereignisse beginnt in dem gerade modalen *View*-Objekt. In der Regel ist das die Instanz von *TApplication*, auf der Sie Ihr Programm aufbauen. Wenn aber das Programm gerade ein Dialogfenster geöffnet hat, dann ist normalerweise dieses Fenster das modale Objekt. In jedem Fall ist es das modale Objekt, das die einlaufenden Ereignisse auf den Weg schickt. Welchen Weg ein Ereignis dann aber einschlägt, hängt ab vom Wesen dieses Ereignisses.

In diesem Zusammenhang unterscheiden wir drei unterschiedliche Arten von Ereignissen: *Positionsereignisse*, *fokussierte Ereignisse* und *Rundrufe*.

Positionsereignisse

sind fast durchweg Mausereignisse (*evMouse*). Das modale *View*-Objekt, dem ein solches Ereignis übermittelt wird, durchsucht die zu ihm gehörenden Subviews, um diejenige herauszufinden, in deren Bereich das Ereignis auftrat. Bei dieser Suche werden die Subviews in der durch die Z-Ordnung bestimmten Reihenfolge abgefragt. Dies ist deshalb so wichtig, weil *View*Objekte einander auf dem Bildschirm überlappen können, also möglicherweise mehr als ein Objekt an der gesuchten Position zu finden ist. Die Einhaltung der Z-Ordnung stellt sicher, daß das Objekt, das auf dem Bildschirm vor allen anderen Objekten dargestellt ist, als erstes dieses Ereignis zu sehen bekommt. Wird ein solches Objekt gefunden, erhält es das Ereignis zur weiteren Bearbeitung. Die Suche beginnt erneut, wenn zu diesem Objekt weitere Subviews gehören.

Dieser Prozeß wird so lange fortgesetzt, bis schließlich ein Objekt kein weiteres Objekt findet, an das es das Ereignis weitergeben könnte. Dann ist möglicherweise dieses Objekt ein terminales *View*-Objekt, eines also, zu dem keine weiteren Subviews gehören. In diesem Fall hat das Ereignis seinen Adressaten, das zuständige Objekt erreicht. Vielleicht wird aber die Position, an der das Ereignis auftrat, von keinem *View*-Objekt verwaltet und damit kann auch kein zuständiges Objekt gefunden werden.

Fokussierte Ereignisse

sind im allgemeinen entweder Tastenanschläge (*evKeyDown*) oder Ereignisse des Typs *evCommand*. Den Weg, den sie dann nehmen, haben wir in Kapitel 5, »View-Objekte«, als Selektierungspfad bezeichnet.

Das modale Objekt nimmt das Ereignis entgegen und gibt es an seine selektierte Subview weiter. Gehören zu dieser weitere Subviews, wird das Ereignis der gerade selektierten übermittelt. Dieser Vorgang wiederholt sich so lange, bis endlich das fokussierte *View*-Objekt gefunden ist. Dieses fokussierte Objekt muß nun auf das Ereignis reagieren.

Kann das Objekt mit dem Ereignis nichts anfangen, schickt es dieses Ereignis wieder auf dem Selektierungspfad zurück zu der Gruppe, zu der es gehört. Auch dieser Prozeß wiederholt sich, bis entweder ein Objekt sich für das betreffende Ereignis zuständig erklärt oder bis dieses Ereignis wieder bei dem modalen Objekt eintrifft, von dem aus es sich auf den Weg machte. Kann

das modale Objekt nun auch wieder nichts mit dem Ereignis anfangen, ruft es die Methode *EventError* auf.

Betrachten wir diesen Vorgang einmal anhand der integrierten Entwicklungsumgebung von Turbo Pascal, wobei wir davon ausgehen, daß mehrere Editor-Fenster geöffnet sind. Sie drücken eine Taste und Sie wissen natürlich genau, in welche der gerade offenen Dateien das betreffende Zeichen eingefügt werden soll. Betrachten wir, wie Turbo Vision sicherstellt, daß das Zeichen auch wirklich dorthin gelangt.

Als Sie die Taste drückten, wurde ein Ereignis des Typs *evKeyDown* erzeugt und dem gerade modalen Objekt, dem *TApplication*-Objekt, übermittelt. *TApplication* schickt dieses Ereignis an seine selektierte Subview weiter, und das ist bei *TApplication* immer ein Objekt des Typs *TDeskTop*, die Arbeitsfläche also. Zu dieser Arbeitsfläche gehören – davon sind wir ausgegangen – mehrere Editorfenster. Das gerade selektierte Editorfenster, also das, das mit einer doppelten Linie umrahmt ist, erhält nun dieses Ereignis. Aber auch solch ein Fenster ist eine Gruppe, zu der ein Rahmen, ein Fensterinhalt und zwei Rollbalken gehören. Unter diesen ist nur das Fensterinnere selektierbar, und da zu diesem keine weiteren Subviews gehören, ist dieses Objekt auch das fokussierte Objekt. Es ist damit zuständig für die Bearbeitung des Tastaturereignisses.

Rundrufe

sind Ereignisse des Typs *evBroadcast* und wir nennen sie Rundrufe, weil sie im Gegensatz zu Positions- oder fokussierten Ereignissen keinen spezifischen Adressaten haben. Solche Ereignisse werden daher allen Subviews übermittelt, die zu dem modalen Objekt gehören.

Auch bei der Übermittlung eines Rundrufs an die Subviews hält sich das modale Objekt an die durch die Z-Ordnung festgelegte Reihenfolge. Ist eine der Subviews eine Gruppe, übermittelt auch sie das Ereignis ihren Subviews, wieder in der Reihenfolge der Z-Ordnung. Dies geschieht so lange, bis schließlich alle direkt oder indirekt zum modalen Objekt gehörenden *View*-Objekte von diesem Ereignis Kenntnis erhalten haben.

Im allgemeinen kommunizieren über solche Rundrufe Objekte miteinander. Wenn Sie zum Beispiel den Rollbalken eines Fensters, das zur Ausgabe einer Textdatei dient, anklicken, muß der Rollbalken dem *TScroller*-Objekt irgendwie mitteilen, daß es einen anderen Teil des Textes darstellen soll. Es sendet also solch einen Rundruf aus, der etwa besagt: »Auf mir ist die Positionsmarkierung verschoben worden!«. Andere Objekte empfangen diesen Rundruf und veranlassen die notwendigen Aktionen. Auf die Einzelheiten einer solchen Kommunikation zwischen Objekten gehen wir später noch ein.

Zuständigkeiten

Jedes *View*-Objekt besitzt das Feld *EventMask*, dessen einzelne Bits bestimmen, für welche Arten von Ereignissen das Objekt zuständig ist. Die Bits in *EventMask* haben die gleiche Bedeutung wie die in *TEvent.What*, weshalb Sie

die schon diskutierten *evXXXX*-Konstanten auch dazu benutzen können, diesem Feld den passenden Wert zuzuweisen. Ist ein Bit in *EventMask* gesetzt, akzeptiert das Objekt das entsprechende Ereignis zur weiteren Bearbeitung. Ist das Bit aber gelöscht, ignoriert das *View*-Objekt diese Art von Ereignissen.

Phase: Umleitung von Ereignissen

Manchmal soll ein anderes *View*-Objekt als das fokussierte die fokussierten Ereignisse (insbesondere Eingaben über die Tastatur) abarbeiten. Nehmen wir beispielsweise ein Textfenster: Sie möchten den darin dargestellten Text rollen und zwar mit den Cursortasten. Die Aufgabe, den Text zu rollen, erledigen die Rollbalken. Da aber das Fenster das fokussierte Objekt ist, bekommen die Rollbalken diese Tastaturereignisse überhaupt nicht zu sehen.

Auch für dieses Problem bietet Turbo Vision eine Lösung, die sicherstellt, daß einem anderen als dem fokussierten Objekt solche Ereignisse zur Bearbeitung übergeben werden. Was wir vorhin zu dem Weg sagten, den fokussierte Ereignisse nehmen, bleibt weiterhin richtig. Wir müssen aber auf zwei Ausnahmen hinweisen, bei denen Ereignisse von dem beschriebenen Selektierungspfad abweichen.

Wenn einem modalen *View*-Objekt ein fokussiertes Ereignis zur Bearbeitung übergeben wird, können wir den Weg, den dieses Ereignis nimmt, in drei Phasen einteilen:

- Das Ereignis wird in der Reihenfolge der Z-Ordnung jeder Subview übergeben, in deren Feld *Option* das Bit *ofPreProcess* gesetzt ist.
- Wurde das Ereignis von keiner dieser Subviews als abgearbeitet markiert, wird es dem fokussierten Objekt übergeben.
- Ist das Ereignis noch immer nicht bearbeitet worden, wird es, wiederum in der Reihenfolge der Z-Ordnung, jeder Subview übergeben, in deren Feld *Option* das Bit *ofPostProcess* gesetzt ist.

Wenn also, wie im genannten Beipiel, der Rollbalken die Tastaturereignisse bearbeiten soll, die an sich zu dem Fensterobjekt gehen, dann muß bei der Initialisierung des Rollbalkens in dessen Feld *Option* das Bit *ofPreProcess* gesetzt werden. In diesem speziellen Fall macht es keinen großen Unterschied, ob Sie das Bit *ofPreProcess* oder das Bit *ofPostProcess* setzen. Das Resultat ist das gleiche. Da das fokussierte Objekt das Ereignis nicht bearbeitet (*TScroller* gibt sich mit Tastaturereignissen nicht ab), wird es dem Rollbalken auf jeden Fall übergeben, entweder bevor oder nachdem das fokussierte Objekt dieses Ereignis zur Kenntnis genommen hat.

Im allgemeinen werden Sie jedoch in dem gegebenen Fall besser das Bit *ofPostProcess* setzen, da Sie dadurch an Flexibilität gewinnen. Vielleicht soll später einmal das Objekt im Fensterinnern Tastaturereignisse entgegennehmen und bearbeiten. Das wäre nicht möglich, wenn der Rollbalken sich vorher alle diese Ereignisse schnappt, bevor sie an das fokussierte Objekt gehen (*ofPreProcess* gesetzt).

Das Feld Phase

Jedes *TGroup*-Objekt definiert ein Feld *Phase*, das einen der drei Werte *phFocused*, *phPreProcess* und *phPostProcess* annehmen kann. Über dieses Feld kann ein *View*-Objekt ermitteln, ob ein ankommendes Ereignis seinen Weg über ein Objekt nahm, in dessen *Options*-Feld das Bit *ofPreProcess* oder *ofPostProcess* gesetzt war oder ob es auf dem Weg zum fokussierten Objekt ist. Diese Unterscheidung ist manchmal notwendig, denn einige *View*-Objekte nehmen – abhängig von Phase – unterschiedliche Ereignisse an oder reagieren auf die gleichen Ereignisse unterschiedlich.

Betrachten wir den Fall eines einfachen Dialogfensters, das ein Eingabefeld und einen mit »***A**ll right*« beschrifteten Aktionsschalter enthält. Für diesen Aktionsschalter ist das Tastenkürzel [A] definiert. Normalerweise müssen Sie sich bei Dialogelementen um den Wert des Feldes *Phase* nicht kümmern. Bei der Initialisierung dieser Elemente wird automatsich das Bit *ofPostProcess* gesetzt. Dadurch werden Tastatureingaben, also fokussierte Ereignisse, an sie weitergegeben, und sie übernehmen den Fokus, wenn der Benutzer das für sie definierte Tastenkürzel betätigt. Drücken Sie [A] und der mit »***A**ll right*« beschriftete Aktionsschalter wird fokussiert.

Aber nehmen wir an, das Eingabefeld sei fokussiert, und Sie drücken [A]. Dann wird das Zeichen »A« in die Eingabezeile eingefügt. Der Aktionsschalter sieht dieses Ereignis nicht, da es bereits von dem Eingabefeld bearbeitet wurde. Im Feld *Option* des Aktionsschalters das Bit *ofPreProcess* zu setzen, nützt in diesem Fall nichts, denn dann würde immer der Schalter dieses Ereignis bearbeiten, und Sie könnten in die Eingabezeile nie ein »A« einfügen.

Die Lösung für dieses Problem ist einfach: Lassen Sie den Aktionsschalter die Ereignisse überprüfen, bevor und nachdem das fokussierte Objekt diese Ereignisse sieht. Voreingestellt ist, daß ein Aktionsschalter ein Ereignis daraufhin überprüft, ob ein Tastenkürzel der Form [Alt][Buchstabe] gedrückt wurde, bevor es an das fokussierte Objekt weitergereicht wird. Das ist der Grund dafür, daß Sie in einem Dialogfenster immer Tastenkürzel der Form [Alt][Buchstabe] benutzen können, Tastenkürzel der Form [Buchstabe] aber nur dann, wenn das fokussierte Objekt dieses Ereignis nicht vorher »wegschnappt«.

Gemäß Voreinstellung ist bei Aktionsschaltern sowohl das Bit *ofPreProcess* als auch das Bit *ofPostProcess* gesetzt. Ihnen werden damit Ereignisse übermittelt bevor und nachdem das fokussierte Objekt diese Ereignisse gesehen hat. Aber die *HandleEvent*-Methode eines Aktionsschalters überprüft Ereignisse nur dann, wenn das fokussierte Objekt dieses Ereignisses schon zur Kenntnis genommen hat:

```
evKeyDown:                    { Teil einer case-Anweisung }
  begin
    C := HotKey(Title^);
    if (Event.KeyCode = GetAltCode(C)) or
      (Owner^.Phase = phPostProcess) and (C <> #0) and
        Upcase(Event.CharCode) = C) or
      (State and sfFocused <> 0) and (Event.CharCode = ' ') then
    begin
```

```
        PressButton;
        ClearEvent(Event);
      end;
    end;
```

Befehle

Die meisten Positions- und fokussierten Ereignisse werden von dem Objekt, das sie abarbeitet, in Befehlsereignisse übersetzt. Das bedeutet, daß ein Objekt auf einen Mausklick oder einen Tastendruck dadurch antwortet, daß es ein Ereignis des Typs *evCommand* erzeugt.

Sie bewirken beispielsweise die Erzeugung eines Positionsereignisses, wenn Sie die Statuszeile der integrierten Entwicklungsumgebung mit der Maus anklicken. Das Programm stellt fest, daß die Position dieses Mausklicks in einem Bereich liegt, für den die Statuszeile verantwortlich ist, und übergibt das Ereignis diesem Objekt.

Dieses wiederum ermittelt, welche Option für den Bereich, in dem der Mausklick stattfand, zuständig ist. Mit dieser Option ist auch ein entsprechender Befehl verbunden, und diesen Befehl speichert die Statuszeile in einem *TEvent*-Record ab und setzt dessen Feld *What* auf den Wert *evCommand*. Das Feld *What* des Records, der das Mausereignis übermittelte, wird dann noch auf *evNothing* gesetzt. Dadurch ist sichergestellt, daß der neu erzeugte *TEvent*-Record, der das Befehlsereignis speichert, von *GetEvent* als nächstes gültiges Ereignis akzeptiert wird.

Befehle definieren

Turbo Vision definiert eine Menge numerischer Konstanten, die dazu dienen, Befehle über Namen zu identifizieren. Diese Befehle werden zum Beispiel mit Menüoptionen verbunden. Aktiviert der Benutzer diese Option, wird der Befehl – wie gesagt, ein numerischer Wert – schließlich der Methode *HandleEvent* übergeben, die entscheidet, welche weiteren Routinen aufzurufen sind, damit die mit der Option spezifizierte Aktion ausgeführt wird.

Wenn wir also jetzt davon sprechen, wie Befehle zu definieren seien und welche Beschränkungen Sie dabei beachten müssen, dann meinen wir damit die Festlegung weiterer solcher Konstanten, wie sie auch Turbo Vision schon definiert, und nicht etwa die Aktionen, die Ihr Programm als Antwort auf die Aktivierung eines Befehls durchführen muß. Einen solchen Befehl festzulegen, ist recht einfach. Sie müssen nur eine Konstante definieren:

```
const
  cmTuMalNichts = 100;
```

Welche Namen Sie Ihren Befehlen geben, ist Ihnen überlassen. Turbo

Vision selbst folgt der Konvention, jeden Befehlsnamen mit dem Kürzel *cm* beginnen zu lassen.

Für eigene Zwecke reserviert Turbo Vision die Bereiche von 0 bis 99 und von 256 bis 999. Ihre Programme können Befehle in den Bereichen 100 bis 255 und 1000 bis 65535 definieren. Warum reserviert Turbo Vision nicht gleich einen zusammenhängenden Bereich? Weil nur die Befehle im Bereich von 0 bis 255 deaktiviert werden können. Davon beansprucht Turbo Vision einige für sich, und darüber hinaus auch einige derjenigen, die nicht deaktiviert werden können. Die folgende Tabelle führt noch einmal die einzelnen Bereiche auf:

Bereich	Reserviert	Deaktivierung möglich
0..99	Ja	Ja
100..255	Nein	Ja
256..999	Ja	Nein
1000..65535	Nein	Nein

Befehle deaktivieren

Manchmal soll der Benutzer bestimmte Befehle zeitweise nicht aktivieren können. Sind zum Beispiel keine Fenster geöffnet, dann macht es wenig Sinn, wenn der Benutzer den Befehl, der ein Fenster schließt (*cmClose*), anwählen kann. Turbo Vision ermöglicht Ihnen deshalb, ganze Befehlsgruppen zeitweise außer Kraft zu setzen und sie später wieder verfügbar zu machen.

Eine solche Befehlsgruppe speichern Sie in einer Variablen des Typs *TCommandSet*. Dieser Datentyp ist definiert als *Set* ***of*** *0..255*, woraus sich auch ergibt, daß nur Befehle im Bereich von 0 bis 255 deaktiviert werden können. Im folgenden Ausschnitt wird eine Grupppe von fünf fensterbezogenen Befehlen deaktiviert:

```
var
  WindowCommands: TCommandSet;
begin
  WindowCommands := (cmNext, cmPrev, cmZoom, cmResize, cmClose);
  DisableCommands(WindowCommands);
end;
```

Ereignisse und wie sie behandelt werden

Wenn Sie nun einen Befehl und darüber hinaus auch ein Dialogelement, über das dieser Befehl zu aktivieren ist, definiert haben, müssen Sie noch eine Verbindung herstellen zwischen diesem Befehl und der Programmaktion, die

diesem Befehl entspricht. Wie schon erwähnt, wird diese Verbindung über die *HandleEvent*-Methode hergestellt.

Jedes *View*-Objekt besitzt eine solche Methode, und sie ist bereits in der Lage, die meisten Eingaben des Benutzers zu bearbeiten. Wenn Sie von einem solchen Objekt erwarten, es solle eine für Ihr Programm spezifische Aktion ausführen, müssen Sie diese *HandleEvent*-Methode überschreiben und in der neuen Methode festlegen, wie das Objekt auf den neuen Befehl reagieren muß.

Die *HandleEvent*-Methode eines Objekts legt dessen Verhalten fest. Besitzen zwei Objekte die gleiche *HandleEvent*-Methode, dann reagieren sie auch in der gleichen Weise auf Ereignisse. Wenn Sie einen neuen Objekttyp ableiten, dann wollen Sie wahrscheinlich, daß sich dieses Objekt – von einigen Änderungen und zusätzlichen Eigenschaften abgesehen – so verhält wie sein Vorfahre. Am einfachsten erreichen Sie dies, wenn Sie innerhalb der neuen *HandleEvent*-Methode die gleichnamige Methode des Vorfahren aufrufen.

Die folgenden Programmzeilen zeigen, wie eine solche *HandleEvent*-Methode im Kern aussehen muß:

```
procedure NachKomme.HandleEvent(var Event: TEvent);
begin
  { hier ist der Platz f¡r Anweisungen,
    die das Verhalten des Vorfahren /ndern }
  Vorfahr.HandleEvent(Event);
  { hier ist der Platz f¡r Anweisungen,
    die zus/tzliche Funktionen ausf¡hren }
end;
```

Wollen Sie also, daß Ihr neues Objekt auf einige Ereignisse anders reagiert, als es der Vorfahre dieses Objekts tat, müssen Sie diese Ereignisse abfangen, *bevor* Sie die *HandleEvent*-Methode des Vorfahren aufrufen. Wenn Ihr Objekt über die geerbten hinaus zusätzliche Funktionen ausführen soll, müssen Sie die entsprechenden Anweisungen *nach* dem Aufruf der *HandleEvent*-Methode des Vorfahren einfügen.

Der TEvent-Record

Wir haben bis jetzt in ziemlich abstrakter Weise über Ereignisse und deren verschiedene Arten gesprochen. Wir haben auch gezeigt, welche Wege sie in Turbo Vision einschlagen müssen, haben uns aber bisher darum herumgedrückt, auf die konkrete Struktur des Records einzugehen, der in Turbo Vision als Transportmittel für solche Ereignisse eingesetzt wird. Das wollen wir jetzt nachholen. Dieser Record ist in der Unit DRIVERS.TPU folgendermaßen definiert:

```
TEvent = record
  What: Word;
  case Word of
```

```
      evNothing: ();
      evMouse: (
        Buttons: Byte;
        Double: Boolean;
        Where: TPoint);
      evKeyDown: (
        case Integer of
    0: (KeyCode: Word);
        1: (CharCode: Char;
            ScanCode: Byte));
      evMessage: (
        Command: Word;
        case Word of
          0: (InfoPtr: Pointer);
          1: (InfoLong: Longint);
          2: (InfoWord: Word);
          3: (InfoInt: Integer);
          4: (InfoByte: Byte);
          5: (InfoChar: Char));
  end;
```

Sie sehen, TEvent ist ein varianter Record. Was wirklich in einem solchen Record gespeichert ist, hängt von dem Wert des Feldes *What* ab. Wenn also dieses Feld den Wert *evMouseDown* enthält, enthält *TEvent* darüber hinaus auch:

```
Buttons: Byte;
Double: Boolean;
Where: TPoint;
```

Speichert *TEvent.What* den Wert *evKeyDown*, können Sie auf diese Daten zugreifen über

```
KeyCode: Word;
```

oder über

```
CharCode: Char;
ScanCode: Byte;
```

Den abschließenden varianten Record bezeichnen wir als das *Informationsfeld*. Es ist in der Lage, ein Datum vom Typ Pointer, LongInt, Word, Integer, Byte oder Char aufzunehmen.

Dieses Informationsfeld wird in Turbo Vision auf verschiedenste Weise genutzt. Wie wir gesehen haben, können *View*-Objekte selbst Ereignisse generieren und an andere Objekte schicken, wobei oft über dieses Feld notwendige Daten mit übermittelt werden. Auf die so vermittelte Kommunikation zwischen Objekten gehen wir später noch näher ein.

Erledigte Ereignisse

Ein Beispiel dafür, wie dieses Informationsfeld eines *TEvent*-Records genutzt wird, bietet der Fall eines schon abgearbeiteten Ereignisses. Das Objekt, das auf dieses Ereignis reagiert hat, ruft die global verfügbare Prozedur *ClearEvent* auf, die das Feld *TEvent.What* auf den Wert *evNothing* setzt und im Informationsfeld *TEvent.InfoPtr* den Wert *@Self* einträgt. Dadurch wird nicht nur das Ereignis als »erledigt« gekennzeichnet, es kann darüber hinaus auch ermittelt werden, welches Objekt das Ereignis bearbeitet hat.

Liegengelassene Ereignisse

Ein Ereignis wird im Normalfall immer ein Objekt finden, das sich mit ihm befaßt. Wenn aber der Fall eintreten sollte, daß kein Objekt sich für zuständig hält, ruft das modale *View*-Objekt die Methode *EventError* auf. Diese wiederum ruft die gleichnamige Methode der Gruppe auf, zu der dieses Objekt gehört. Das geht so weiter, bis endlich *TApplication.EventError* erreicht wird. Und diese Methode tut, so wie sie in Turbo Vision definiert ist, überhaupt nichts.

Sie können sie überschreiben, um etwa eine entsprechende Fehlermeldung auszugeben, wenn dies sinnvoll ist. Oder sie sind vielleicht ein Anhänger piepsender Programme, dann können Sie über diese Methode ein solches Geräusch erzeugen lassen. Vergessen Sie aber in diesem Fall nicht, in Ihrem Programm ein Dialogfenster vorzusehen, das es den Benutzern Ihres Programms ermöglicht, dieses »stummzuschalten«, denn möglicherweise teilt ja ein Benutzer Ihre Vorlieben nicht. Turbo Vision jedenfalls stellt Ihnen das zu diesem Zweck geeignete Objekt zur Verfügung: ein *TRadioButton*.

Den Event-Mechanismus verändern

Den Angelpunkt eines modalen Objekts bildet eine Schleife, die – wenn wir alle Einzelheiten, die uns hier nur stören würden, weglassen – aussieht, wie im folgenden Ausschnitt gezeigt:

```
var E: TEvent;

E.What := evNothing;
repeat
  if E.Whatev <> Nothing then EventError(E);
  GetEvent(E);
  HandleEvent(E);
until EndState <> Continue;
```

Zentrale Sammelstelle

Einer der größten Vorzüge, den ereignisgesteuerte Programme bieten, liegt darin, daß Sie nicht berücksichtigen müssen, woher die Ereignisse kommen. Ein *TWindow*-Objekt beispielsweise schließt das Fenster, wenn ihm als Ereignis der Befehl *cmClose* übermittelt wird. Es kümmert sich nicht darum, ob dieses Ereignis generiert wurde, weil der Benutzer das Schließfeld anklickte, das entsprechende Tastenkürzel benutzte oder gar ein anderer Teil des Programms diesen Befehl ausgesendet hat. Es kümmert sich nicht einmal darum, ob dieser Befehl überhaupt für dieses Objekt bestimmt war. Aber es weiß, daß es zuständig ist für Ereignisse dieser Art und auf sie zu reagieren versteht, also wird das Fenster geschlossen.

Die Beschaffung von Ereignissen ist einzig und allein Aufgabe der *GetEvent*-Methode eines Turbo-Vision-Programms. Und nur diese Methode kümmert sich um die Herkunft von Ereignissen. Die Objekte Ihres Programms rufen einfach *GetEvent* auf und verlassen sich darauf, daß diese Methode die Maus und die Tastatur abfragt, und auch solche Ereignisse entgegennimmt, die von anderen Teilen des Programms ausgesendet wurden.

Nehmen wir nun an, Sie wollten eine neue Art von Ereignissen bearbeiten lassen, deren Quelle die serielle Schnittstelle sei. Dazu müssen Sie einfach die Methode *TApplication.GetEvent* überschreiben. Diese Methode ruft ihrerseits *TProgram.GetEvent* auf, und Sie können in APP.PAS nachlesen, daß *TProgram.GetEvent* zunächst die Maus, dann die Tastatur abfragt und schließlich die Methode *Idle* aufruft. Daraus ergeben sich für Sie zwei Wege, Ihr Vorhaben zu realisieren. Sie können die Methode *Idle* überschreiben, damit sie die serielle Schnittstelle abfragt. Oder Sie können eine Routine schreiben – nennen wir sie *GetComEvent* –, die das gleiche tut, und dann *GetEvent* selbst überschreiben. In der neuen Methode rufen Sie *GetComEvent*(*Event*) auf, und diese Routine gibt ein Ereignis zurück, sollte ein solches an der seriellen Schnittstelle verfügbar sein.

GetEvent überschreiben

Die Methode *GetEvent* des aktuellen modalen Objekts ruft die gleichnamige Methode der Gruppe auf, zu der dieses Objekt gehört. Und auch dies wiederholt sich so lange, bis endlich *TApplication.GetEvent* aufgerufen wird, die dann tatsächlich das nächste erreichbare Ereignis einsammelt. Überschreiben Sie diese eine Methode, und Sie ändern die Ereignisbearbeitung für Ihr ganzes Programm:

```
procedure TMyApp.GetEvent(var Event: TEvent);
begin
  TApplication.GetEvent(Event);
  { Anweisungen daf¡r, was sonst noch
```

```
    als Ereignis gelten soll }
end;
```

Damit haben Sie es zum Beispiel in der Hand, Ihr Programm mit einer Makrofunktion zu versehen, die Eingaben aufzeichnet und diese Aufzeichnungen zu einem späteren Zeitpunkt wieder abspielt. Wie schon gesagt, kümmern sich die Objekte nicht darum, woher die Ereignisse kommen. Wenn ein Objekt für ein Ereignis zuständig ist und weiß, wie es auf dieses Ereignis zu reagieren hat, dann reagiert es auch. Punkt.

Unproduktive Zeiten nutzen

Je leistungsfähiger Computer werden, desto schneller warten sie auf die Eingaben des Benutzers. Mit Turbo Vision können Sie wenigstens einen Teil dieser unproduktiven Zeiten nutzen, denn *GetEvent* ruft, wie wir gesehen haben, die Methode *Idle* auf. So wie sie definiert ist, tut diese Methode gar nichts. Sie müssen sie also überschreiben:

```
procedure TMyApp.Idle
begin
  HeapViewer.Update;
end;
```

Der obige Programmausschnitt nimmt an, Sie hätten ein Objekt *THeapView* und dessen Methode *Update* definiert. Dadurch sieht der Benutzer immer, wieviel Arbeitsspeicher noch frei ist, gleichgültig in welchem Teil des Programms er sich gerade befinden mag.

Kommunikation zwischen Objekten

Objekte bilden eine Einheit aus Daten und Methoden, die mit diesen Daten arbeiten. Nun nehmen wir an, daß innerhalb Ihres Programms Objekte Informationen miteinader austauschen müßten. In einem traditionellen Programm würde das einfach bedeuten, daß Informationen aus einer Datenstruktur zu kopieren und in eine andere zu übertragen sind. In einem objektorientierten Programm ist das aber nicht so leicht.

Kommunikation zwischen zwei *View*-Objekten ist nicht so einfach herzustellen, wie Daten zwischen zwei Programmkomponenten hin- und herzukopieren. Wenn eine solche Kommunikation notwendig wird, sollten Sie sich zunächst fragen, ob Sie die Aufgaben, die diese Objekte erledigen, zwischen beiden richtig aufgeteilt haben. Vielleicht ist ja nur ein falscher Programmentwurf die Ursache des Problems. Wenn Sie die Eigenschaften beider Objekte in einem neuen Objekt zusammenfassen, ist das Problem möglicherweise schon gelöst.

Möglich ist aber auch, daß das Programmdesign völlig in Ordnung ist und dennoch die Notwendigkeit besteht, daß Objekte Daten austauschen. Dann kann die Entwicklung eines eigenen Objekts, das genau die Aufgabe des Datenaustauschs übernimmt, ein möglicher Ausweg sein.

Nehmen wir an, Sie hätten ein Objekt Tabellenkalkulation und ein Objekt Textverarbeitung und wollen Daten aus der Tabellenkalkulation in die Textverarbeitung übernehmen und umgekehrt. Sie könnten Ihre Objekte so verändern, daß beide eine Schnittstelle definieren, über die sie Daten aus dem anderen Objekt übernehmen und dem anderen Objekt übergeben können. Aber zu einem späteren Zeitpunkt gesellt sich zu diesen beiden Objekten das Objekt Datenbank. Auch aus diesem sollen Daten übernommen werden können, auch dieses soll in der Lage sein, Daten der anderen Objekte anzufordern.

Zwischenablage

Sie sehen, der bessere Weg besteht darin, die Aufgabe des Datenaustauschs einem eigenen Objekt zu übertragen, in diesem Fall dem Objekt Zwischenablage. Ein anderes Objekt braucht dann nur zu wissen, wie es Daten an dieses Objekt übergibt und wie es Daten übernimmt. Es braucht sich weder darum zu kümmern, wohin die abgelieferten Daten gehen, noch darum woher die übernommenen Daten stammen.

Unter uns Objekten gesagt...

Nun, Sie wissen, daß der Entwurf Ihres Programms in Ordnung ist, und daß auch ein Objekt für den Datenaustausch nicht in Frage kommt, denn das hieße, ein Spatzenproblem mit Kanonen erledigen zu wollen. Sie müssen also einen einfachen Informationsaustausch zwischen zwei *View*-Objekten implementieren.

Bevor ein Objekt mit einem anderen kommunizieren kann, muß es herausbekommen, wo sich dieses andere Objekt befindet, und vielleicht auch, ob zum gegebenen Zeitpunkt dieses andere Objekt überhaupt existiert. Dazu zunächst ein einfaches Beispiel.

Die Unit *StdDlg* enthält das Objekt *TFileDialog*, über das Sie in der integrierten Entwicklungsumgebung eine neue Datei laden. Zu diesem Objekt gehört ein Objekt *TFileList*, das eine Liste der Dateien im aktuellen Verzeichnis darstellt, und das Objekt *FileInputLine*, das den Namen der Datei ausgibt, auf der sich der Leuchtbalken innerhalb der Liste gerade befindet. Versetzt der Benutzer diesen Leuchtbalken, muß das Objekt *TFileList* dem Objekt *TFileInputLine* den Namen dieser Datei mitteilen.

In diesem Fall kann *TFileList* sicher sein, daß *TFileInputLine* existiert, denn beide Objekte wurden über die Initialisierung von *TFileDialog* auch initiali-

siert. Wie teilt nun das Objekt *TFileList* dem Objekt *TFileInputLine* mit, daß der Benutzer einen anderen Dateinamen angewählt hat?

Einfach dadurch, daß *TFileList* eine entsprechende Botschaft abschickt. Der folgende Ausschnitt zeigt *TFileList.FocusItem*, das dieses Ereignis ausendet, und *TFileInputLine.HandleEvent*, das diese Botschaft empfängt.

```
procedure TFileList.FocusItem(Item: Integer);
begin
  TSortedListBox.FocusItem(Item);
  Message(TopView, evBroadcast, cmFileFocused, List^.At(Item));
  { TopView zeigt auf das aktuell modale View-Objekt }
end;

procedure TFileInputLine.HandleEvent(var Event: TEvent);
var
  Dir: DirStr;
  Name: NameStr;
  Ext: ExtStr;
begin
  TInputLine.HandleEvent(Event);
  if (Event.What = evBroadcast) and (Event.Command = cmFileFocused)
     and  (State and sfSelected = 0) then
  begin
     if PSearchRec(Event.InfoPtr)^.Attr and Directory <> 0 then
        Data^ := PSearchRec(Event.InfoPtr)^.Name + '\'+
          PFileDialog(Owner)^.WildCard
     else Data^ := PSearchRec(Event.InfoPtr)^.Name;
     DrawView;
  end;
end;
```

Beachten Sie, daß *TFileList.FocusItem* die erweiterte Syntax von Turbo Pascal (einzuschalten über den Compilerschalter **$X+**) benutzt, um die Funktion *Message* wie eine Prozedur zu verwenden, denn diese Methode interessiert sich nicht dafür, was diese Funktion zurückliefert.

Wer hört die Botschaft?

Nehmen wir an, Sie müßten zunächst ermitteln, ob gerade ein Fenster geöffnet ist, bevor Ihr Programm weitere Aktionen durchführen kann. Wie kann Ihr Programm das herausfinden? Nun, es muß eine Botschaft aussenden, die ein solches Fenster versteht. Wir haben schon erwähnt, daß ein Objekt, das ein Ereignis abgearbeitet hat, in dem Ereignis-Record eine »Unterschrift« ablegt. Ihr Programm kann also sehr leicht erfahren, welches Objekt, wenn überhaupt eines, auf dieses Ereignis reagiert hat.

Greifen wir wieder auf die integrierte Entwicklungsumgebung zurück. Wenn der Benutzer ein Watch-Fenster öffnen will, muß sich das Programm

zunächst vergewissern, daß ein solches Fenster nicht schon geöffnet ist. Sollte das der Fall sein, muß es im Vordergrund des Bildschirms dargestellt, andernfalls geöffnet werden. Eine solche Botschaft auszusenden ist einfach:

```
AreYouThere := Message(DeskTop, evBroadcast, cmYoWindow, nil);
```

Innerhalb der *HandleEvent*-Methode des Watch-Fensters reagiert eine Anweisung auf die ausgesendete Botschaft. Sie veranlaßt mittels *ClearEvent*, daß dieses Ereignis als erledigt markiert wird:

```
case Event.Command of
  ...
  cmYoWindow: ClearEvent(Event);
  ...
end;
```

Wie wir wissen, setzt *ClearEvent* nicht allein das Feld *What* des Ereignisrecords auf *evNothing*, sondern speichert in *InfoPtr* auch den Wert *@Self*. *Message* liest dieses Feld, und gibt – sofern es nicht **nil** ist – einen Zeiger auf das Objekt zurück, welches das Ereignis bearbeitet hat, in diesem Fall also auf das Watch-Fenster. Jedenfalls kann das Programm über den zurückgelieferten Wert entscheiden, was zu tun ist:

```
if AreYouThere = nil then
  CreateWatchWindow          { neues Watch-Fenster }
  else AreYouThere^.Select; { andernfalls das schon vorhandene im
                              Vordergrund neu ausgeben }
```

Solange das Watch-Fenster das einzige Objekt ist, das auf die Botschaft *cmYoWindow* reagiert, kann das Programm, nachdem die oben dargestellten Anweisungen abgearbeitet wurden, davon ausgehen, daß nur ein Watch-Fenster offen ist, und dieses Fenster vor allen anderen Objekten auf dem Bildschirm zu sehen ist.

Hilfe zur Hilfe

Sie kennen die kontext-sensitive Hilfe, die Sie in der integrierten Entwicklungsumgebung über [F1] abrufen können. Turbo Vision selbst stellt kein solches Hilfesystem zur Verfügung, bietet aber die Objekte, die Sie brauchen, um ein Hilfesystem zu entwickeln, und bietet darüber hinaus die Möglichkeit, dieses System in Ihre Programme zu integrieren.

Nehmen wir an, Sie hätten ein Objekt *HelpViewer* entwickelt, dessen Aufgabe darin besteht, Hilfeinformationen auf dem Bildschirm darzustellen. Dieses Objekt basiert auf einem *TStream*-Objekt, und die Einträge in diesem Stream sind mittels eines numerischen Werts als Schlüssel indiziert. Dieser numerische Wert ist ein Hinweis auf den aktuellen Programmkontext, aus dem heraus der Benutzer Hilfe angefordert hat. Turbo Vision sorgt dann dafür, daß Ihr Programm auch dem Hilfesystem den richtigen numerischen Wert

übermittelt, den das System braucht, um den passenden Hilfebildschirm auszugeben.

Sie können jedem *View*-Objekt bei seiner Initialisierung einen solchen numerischen Wert übergeben, und diesen Wert können Sie mittels der Methode *GetHlpCtx* erfragen. Eine besondere Rolle spielt dabei die in Turbo Vision definierte Konstante *hcNoContext*. Nehmen wir an, Sie hätten ein Dialogfenster definiert, das einige Dialogelemente enthält. Sie können nun jedem dieser Elemente und auch dem Dialogfenster selbst eine Hilfenummer übergeben. Sie entscheiden sich aber dafür, einen Hilfetext zu schreiben, der alle Probleme behandelt, die dieser Dialog dem Benutzer stellen mag. Sie sparen sich dadurch separate Hilfetexte für jedes einzelne Element des Dialogs. Nun fokussiert der Benutzer eines dieser Elemente und ruft dann die Hilfestellung ab. Sie wollen natürlich, daß in diesem Fall Ihrem Hilfesystem die Hilfenummer des Dialogfensters und nicht die Hilfenummer des Elements übermittelt wird. Um dies zu erreichen geben Sie den Dialogelementen bei ihrer Initialisierung den Wert *hcNoContext* mit. Wird dann über *GetHlpCtx* die Hilfenummer abgefragt, sorgt Turbo Vision automatisch dafür, daß die Hilfenummer der Gruppe, in diesem Fall also des Dialogfensters, zurückgeliefert wird.

Diese Hilfenummern werden auch von der Statuszeile benutzt, um zu bestimmen, welches *View*-Objekt in der Statuszeile auszugeben ist. Dadurch haben Sie die Möglichkeit, in der Statuszeile zum Bespiel eine Kurzinformation über die Menüoption auszugeben, die der Benutzer gerade angewählt hat.

7. Sichere Programme schreiben

Die Behandlung von Fehlern ist bei einem interaktiven Programm naturgemäß um einiges aufwendiger, als bei einem Programm, das über Parameter in der Kommandozeile gesteuert wird. Tritt hier ein Fehler auf, dann genügt die Ausgabe einer entsprechenden Meldung, und das Programm wird abgebrochen. Solches Verhalten ist durchaus annehmbar – und wird eigentlich auch gar nicht anders erwartet. Ein interaktives Programm dagegen muß einen Fehler abfangen und in einem definierten Zustand weiterarbeiten. Datenverlust wegen eines vorzeitigen Programmabbruchs kann nicht hingenommen werden. Erst ein Programm, das diese Voraussetzung erfüllt, kann als sicher angesehen werden.

Turbo Vision erleichtert das Schreiben solch sicherer Programme dadurch, daß es einen Stil des Programmierens unterstützt, der das Auffangen von Fehlern ermöglicht, insbesondere solchen, die durch mangelnden freien Arbeitsspeicher ausgelöst werden. Erreicht wird dies durch eine Strategie, die auf dem Konzept der atomaren Operationen aufbaut.

Alles oder nichts

Zerteilen Sie eine komplexe Operation in mehrere kleine, die ihrerseits nicht mehr sinnvoll unterteilt werden können, dann erhalten Sie eine Sammlung atomarer Operationen. Bezogen auf unser Problem gilt damit: Eine komplexe Operation ist dann erfolgreich ausgeführt, wenn jede atomare Operation erfolgreich war. Umgekehrt: Schlägt auch nur eine atomare Operation fehl, gilt die Gesamtoperation als gescheitert.

Ein Programm fordert typischerweise sehr oft Speicher in unterschiedlich großen Portionen an. Nehmen wir den Fall, Sie wollten ein Dialogfenster ausgeben. Dazu müssen Sie Speicher anfordern für das Dialogfenster selbst, weiteren Speicher für jedes Eingabeelement, das Sie darstellen wollen. Jede dieser Speicheranforderungen kann natürlich fehlschlagen, weshalb Sie nach jeder Anforderung überprüfen müssen, ob alles in Ordnung ging. War das bei auch nur einer Anforderung nicht der Fall, müssen Sie all den Speicher, der bis dahin den Objekten Ihrer Dialogfenster zugewiesen wurde, wieder freigeben. Sie würden natürlich lieber allen Elementen Ihres Dialogfensters auf

einen Schlag Speicher zuweisen, und dann erst überprüfen, ob diese Operation erfolgreich war. Und hier kommt die *Sicherheitszone* von Turbo Vision ins Spiel.

Die Sicherheitszone

Turbo Vision reserviert einen kleinen Teil des Arbeitsspeichers – in der Voreinstellung vier KByte – am Ende des Heaps; nennen wir diesen Bereich die *Sicherheitszone*. Wenn nun eine Anforderung von Heapspeicher nur dann erfüllt werden kann, wenn auch ein Teil dieses Bereichs bereitgestellt wird, gibt die in Turbo Vision definierte Funktion *LowMemory* True zurück, was anzeigt, daß weitere Speicheranforderungen sehr wahrscheinlich fehlschlagen werden.

Damit dieses Konzept funktioniert, muß die Sicherheitszone ausreichend groß dimensioniert werden: Sie muß so groß sein, daß die Speicheranforderungen zwischen der Abfrage von *LowMemory* erfolgreich sind. Die voreingestellte Größe von vier KByte wird für die meisten Anwendungen ausreichen.

Nutzen wir die überkommenen Techniken der Speicheranforderung, wird der Aufbau eines Dialog-Fensters aussehen wie im folgenden Beispiel:

```
valid:=true;
R.Assign(20,3,60,10);
D := New(Dialog, Init(R, 'Mein Dialog'));
if D <> nil then
begin
  with D^ do
  begin
    R.Assign(2,2,32,3);
    Control := New(PStaticText, Init(R,
      'Sie wollen das wirklich tun?'));
    if Control <> nil then Insert(Control)
    else valid := false;
    R.Assign(5,5,14,7);
    Control := New(PButton, Init(R, '~J~a', cmYes));
    if Control <> nil then Insert(Control)
    else valid := false;
    R.Assign(16,6,25,7);
    Control := New(PButton, Init(R, '~N~ein', cmNo));
    if Control <> nil then Insert(Control)
    else valid := false;
    R.Assign(27,5,36,7);
    Control := New(PButton, Init(R, '~A~bbruch', cmCancel));
    if Control <> nil then Insert(Control)
    else valid := false;
  end;
```

```
  if not valid then Dispose(D, Done);
end;
```

Über die Variable *valid* wird ermittelt, ob eine Speicherzuweisung fehlschlug. In einem solchen Fall wird der von *D* (dem Dialogobjekt) belegte Speicher wieder freigegeben, was – wie wir wissen – bedeutet, daß auch der von Subviews belegte Speicher wieder frei wird. Die Einrichtung der Sicherheitszone ermöglicht uns, den ganzen Block von Anweisungen, die Speicher anfordern, als eine einzige atomare Operation zu behandeln:

```
R.Assign(20,3,60,10);
D := New(Dialog, Init(R, 'Mein Dialog'));
with D^ do
begin
  R.Assign(2,2,32,3);
  Insert(New(PStaticText, Init(R,
    'Sie wollen das wirklich tun?')));
  R.Assign(5,5,14,7);
  Insert(New(PButton, Init(R, '~J~a', cmYes)));
  R.Assign(16,6,25,7);
  Insert(New(PButton, Init(R, '~N~ein', cmNo)));
  R.Assign(27,5,36,7);
  Insert(New(PButton, Init(R, '~A~bbruch', cmCancel)));
end;
if LowMemory then
begin
  Dispose(D, Done);
  OutOfMemory;
  doit := false;
end
else
  doit := Desktop^.ExecView(D) = cmYes;
```

Da die Sicherheitszone weit mehr Speicher belegt, als für den Aufbau der Dialogfenster angefordert werden muß, ist sichergestellt, daß alle Speicheranforderungen erfüllt werden. Erst danach wird *LowMemory* überprüft. Gibt diese Funktion True zurück, wird der belegte Speicherplatz wieder freigegeben, andernfalls die Dialogfenster dargestellt.

Die Überprüfung von *LowMemory* wird sehr oft erfolgen müssen; daher definiert *TApplication* eine Methode – *ValidView* –, die diese Kontrolle ausführt. Der Test am Ende des oben angeführten Programmabschnittes wird bei Nutzung von *ValidView* so formuliert:

```
doit := (ValidView(D) <> nil) and
  (DeskTop^.ExecView(D) = cmYes);
```

ValidView gibt einen Zeiger zurück, der auf **nil** gesetzt ist, sollte das *View*-Objekt ungültig sein. Stellt *ValidView* fest, daß *LowMemory* True zurückgibt, ruft es die in *TApplication* definierte Methode *OutOfMemory* auf.

Andere Fehler

Nun haben leider nicht alle Fehler ihre Ursache in fehlendem Arbeitsspeicher. So kann es sein, daß ein *View*-Objekt Informationen aus einer Datei lesen muß, diese Datei aber das falsche Format hat oder gar nicht existiert. Ein solcher Fehler muß dem Benutzer mitgeteilt werden. Für solche Zwecke ist die Methode *Valid* vorgesehen.

Sofern sie nicht überladen wurde, gibt *TView.Valid* immer True zurück. Dagegen wird die *Valid*-Methode von *TGroup* nur dann True zurückgeben, wenn jede Subview dieser Gruppe True zurückgibt. Auch hier gilt also: Eine Gruppe ist nur dann gültig, wenn jedes Mitglied dieser Gruppe gültig ist. Gibt *Valid* False zurück, darf das betreffende *View*-Objekt nicht benutzt werden, weil zum Beispiel eine ihm zugeordnete Datei nicht geöffnet werden konnte. Eine typische *Valid*-Methode sehen Sie im folgenden Listing:

```
function TMyView.Valid(Command: Word): Boolean;
begin
  Valid := True;
  if Command = 0 then
  begin
    if ErrorEncountered then
    begin
      ReportError;
      Valid := False;
    end;
  end;
end;
```

Wird das *View*-Objekt initialisiert, hat der Parameter *Command* immer den Wert 0. Die Methode *Valid* wird aber auch immer dann aufgerufen, wenn das betreffende *View*-Objekt deaktiviert wird. In diesem Fall erhält *Command* den Wert des Befehls, mit dem das *View*-Objekt deaktiviert wurde (vergleichen Sie dazu Kapitel 5, »View-Objekte«). Wenn Sie also ein Objekt über seinen Konstruktor initialisieren, rufen Sie *ValidView* auf, bevor Sie dieses Objekt benutzen. *ValidView* ruft seinerseits die *Valid*-Methode auf, und stellt darüber hinaus sicher, daß die Sicherheitszone intakt blieb. Gibt die *Valid*-Methode False zurück, dann ist auch der Benutzer über das Auftreten eines Fehlers informiert, denn genau das ist die Aufgabe von *ReportError* im oben gezeigten Beispiel.

Sie können den Aufbau von Bildschirm- oder Dialogfenstern als eine atomare Operation behandeln, wenn Sie sich dabei auf die *Valid*-Methode stützen. Jedes Element, das zu dem Fenster gehört, kann initialisiert werden, ohne daß jedesmal überprüft werden müßte, ob ein Fehler aufgetreten ist. Wenn der Konstruktor eines Objekts seine Arbeit nicht ordnungsgemäß ausführen kann, setzt er *Valid* auf False. Ist das Fenster als Ganzes initialisiert, wird es *ValidView* übergeben. Stellt sich dann heraus, daß die Initialisierung auch nur eines Elementes fehlschlug, wird das ganze Fenster-Objekt als

ungültig angesehen: *ValidView* gibt den von diesem Objekt belegten Speicherplatz wieder frei und einen **nil**-Zeiger zurück. Diesen Zeiger zu überprüfen ist alles, was Sie tun müssen.

Speicherfresser

Die *Valid*-Methode kann auch dort zur Überprüfung eingesetzt werden, wo eine Routine Speicher in einer Größe anfordern muß, die weit über die Größe der Sicherheitszone hinausgeht. Ein solcher Fall tritt zum Beispiel ein, wenn eine Datei vollständig in den Speicher einzulesen ist. Ein derartiger Großverbraucher in Sachen Speicher muß die Überprüfung von *LowMemory* selbst in die Hand nehmen, anstatt sie *ValidView* zu überlassen. Sollte weniger Speicher vorhanden sein, als eine *View* anfordert, ruft sie selbst *Application^.OutOfMemory* auf. Weiterhin muß sie auch ein entsprechendes Flag setzen, das *Valid* überprüfen und gegebenenfalls False zurückgeben kann.

Das Programm FILEVIEW.PAS, mit dessen Hilfe Sie den Inhalt von Textdateien ausgeben können, nutzt die beschriebenen Techniken und erfüllt die Anforderungen an ein sicheres Programm. Sie finden es unter den Demoprogrammen auf den Originaldisketten.

8. Kollektionen

Viel Zeit verwenden Programmierer darauf, Routinen zu schreiben, die Datenstrukturen handhaben, und dieselben Routinen werden mit nur wenigen Veränderungen immer wieder neu geschrieben, neu auf Fehler überprüft, neu von solchen Fehlern befreit.

Pascal ist zweifellos eine mächtige Programmiersprache, und doch liefert sie nur Records und Arrays als vordefinierte strukturierte Datentypen. Alles, was darüber hinausgeht, müssen Sie selbst entwickeln.

Und so schreibt ein Programmierer Routinen, die ein Array initialisieren, Daten in diesem Array speichern, Daten aus einer Datei in dieses Array übertragen, Daten zur Bearbeitung aus diesem Array entnehmen. Dann stellt er fest: sein Programm benötigt einen weiteren, neuen Datentyp. Lesen Sie bitte am Anfang dieses Absatzes weiter.

Wäre es nicht schön, einen Arraytyp zu haben, der die verschiedensten Daten aufnehmen kann, ähnlich einer Kirchenkollekte, die ohne Protest sowohl Geldstücke als auch Hosenknöpfe schluckt? Ein Arraytyp, der die notwendigen Routinen schon mitbringt, die jene Operationen erledigen, die üblicherweise auf Arrays angewendet werden? Routinen, die noch dazu erweitert werden können, ohne daß der Programmcode der schon vorhandenen angetastet werden muß?

Genau das liefert Ihnen das Objekt *TCollection* in Turbo Vision. Es nimmt eine Sammlung generischer Zeiger auf und stellt eine Anzahl Methoden zur Bearbeitung dieser Zeiger zur Verfügung.

Das Objekt TCollection

Neben seiner Eigenschaft, ein Objekt zu sein und der damit verbundenen Annehmlichkeit, auch über eine Reihe von Methoden zu verfügen, überwindet das Objekt *TCollection* zwei Einschränkungen normaler Pascal-Arrays:

Dynamische Anpassung der Größe

Die Größe eines traditionellen Pascal-Arrays muß zum Zeitpunkt der Compilierung feststehen, und das ist dann ganz in Ordnung, wenn Sie genau wissen, wieviel Platz Ihre Daten benötigen werden. Leider ist diese Voraussetzung meist nicht gegeben, weshalb Sie Speicher verschwenden (Array größer als erforderlich) oder die Möglichkeiten Ihres Programms einschränken (Array kleiner als notwendig). Und eine Neudimensionierung Ihres Arrays wird allemal eine Neucompilierung des Programms erforderlich machen.

Dem Objekt *TCollection* übergeben Sie eine Anfangsgröße; während der Laufzeit Ihres Programms paßt sich *TCollection* dynamisch an die Größe der ihm übergebenen Daten an. Insgesamt ein Gewinn an Flexibilität für Ihre Programme.

Polymorphe Daten

Die zweite Einschränkung herkömmlicher Pascal-Arrays besteht darin, daß sie nur Daten gleichen Typs aufnehmen können, und dieser Typ muß zum Zeitpunkt der Compilierung bekannt sein.

Kollektionen umgehen diese Begrenzung durch die Verwendung untypisierter Zeiger. Dies ist nicht allein schnell und effizient, daraus ergibt sich der weitere Vorteil, daß eine solche Kollektion Objekte (oder auch ganz allgemein Daten) verschiedenen Typs und verschiedener Größe aufnehmen kann. Wie ein *Stream* braucht auch eine Kollektion nicht zu wissen, was ihr übergeben wird – sofern es in den Speicher paßt.

Kollektionen und Typüberprüfung

Es sei nicht verschwiegen: Eine Kollektion umgeht die in Pascal traditionell strenge Typüberprüfung. Sie können in einer Kollektion irgend etwas speichern und dann irgend etwas wieder herausnehmen: der Compiler hat keine Möglichkeit zu überprüfen, ob das, was Sie erhalten, das ist, was Sie wollten. Sie können ein *PGeldstueck* abspeichern, dann einen *PHosenknopf* herausnehmen, die Kollektion kann Sie über diesen Mißgriff nicht informieren.

Manchem Pascal-Programmierer mag ein Schauder über den Rücken laufen, ob solchen Tuns. Und tatsächlich erspart diese Typprüfung Stunden um Stunden der Jagd nach nur schwer faßbaren Fehlern. Gehen Sie also mit Umsicht vor: Sie ahnen möglicherweise gar nicht, wie schwer ein solcher Fehler zu finden ist, denn bisher hat der Compiler Ihnen diese Arbeit abgenommen. Wenn jedoch Ihr Programm hängt oder sich in anderer Weise seltsam benehmen sollte, dann überprüfen Sie sorgfältig den Typ der Objekte, die Sie aus einer Kollektion lesen lassen.

Daten in Kollektionen speichern

Wie erwähnt, ist eine Kollektion nicht darauf beschränkt, Objekte aufzunehmen. Sie kann auch ganz »normale« Daten speichern, da eine Kollektion nichts anderes erwartet als einen untypisierten Zeiger. Und das ist Anlaß, ein zweites Mal zur Vorsicht zu mahnen: Einige der Methoden von *TCollection* erwarten, mit Objekten zu arbeiten, die von *TObject* abgeleitet wurden. Dies betrifft insbesondere die Standardmethoden *GetItem* und *PutItem* für den Zugriff auf einen *Stream*, darüber hinaus aber auch die Prozedur *FreeItem*.

Das bedeutet für Sie, daß Sie beispielsweise einen *PString* in einer Kollektion speichern können. Übergeben Sie dann aber diese Kollektion einem *Stream*, wird das Ergebnis alles andere als erfreulich sein, es sei denn, Sie hätten die Methoden *GetItem* und *PutItem* überladen. Wenn Sie andererseits diese Kollektion wieder loswerden wollen, wird deren Destruktor versuchen, den von den einzelnen Einträgen belegten Speicherplatz über *FreeItem* wieder freizugeben. Wenn Sie also normale Daten in einer Kollektion speichern wollen, müssen Sie zumindest die Methoden *GetItem*, *PutItem* und *FreeItem* neu definieren. Genau dies tut zum Beispiel *TStringCollection*, ein Nachkomme von *TCollection*.

Wenn Sie mit Umsicht vorgehen, dann sind Kollektionen und ihre Abkömmlinge, die Sie definieren, schnelle, flexible und verläßliche Datenstrukturen. In Turbo Vision selbst benutzt *TGroup* eine Kollektion zur Verwaltung seiner zugehörigen Elemente.

Eine Kollektion erstellen

ist ebenso einfach wie die Definition des Datentyps, der in der Kollektion zu speichern ist. Nehmen wir an, Sie führen eine beratende Tätigkeit aus und Sie wollen die Kundennummer, den Namen und die Telefonnummer der von Ihnen Beratenen in einer Kollektion speichern. Dazu definieren Sie zunächst das Objekt *TClient* und einen Zeiger auf dieses Objekt:

```
type
  PClient = ^TClient;
  TClient = object(TObject)
    Account, Name, Phone: PString;
    constructor Init(NewAccount, NewName, NewPhone: String);
    destructor Done; virtual;
    procedure Print; virtual;
  end;
```

Darauf implementieren Sie die Methoden *Init* und *Done*, die Speicherplatz für einen Kundenobjekt reservieren bzw. den reservierten Speicher wieder freigeben. Die Felder des Objekts sind alle vom Typ *PString*, ein Zeiger auf einen String. Für diesen muß nur der tatsächlich benötigte Speicher reserviert werden, also die Zahl der Zeichen im String + 1 für das Längenbyte. Die

Funktion *NewStr* berücksichtigt dies, und die Prozedur *DisposeStr* gibt entsprechend auch nur den tatsächlich reservierten Speicher frei:

```
constructor TClient.Init(NewAccount, NewName, NewPhone: String);
begin
  Account := NewStr(NewAccount);
  Name := NewStr(NewName);
  Phone := NewStr(NewPhone);
end;

destructor TClient.Done;
begin
  DisposeStr(Account);
  DisposeStr(Name);
  DisposeStr(Phone);
end;
```

Die Methode *TClient.Done* wird automatisch für jedes in der Kollektion gespeicherten Kundenobjekt aufgerufen, wenn Sie über die Methode *TCollection.Done* diese Kollektion freigeben. Nun können Sie eine Instanz von *TCollection* zur Speicherung ihrer Kundendaten deklarieren und dann die Objekte in die Kollektion einfügen. Der folgende Ausschnitt aus TVGUID17.PAS zeigt dies:

```
var
  ClientList: PCollection;

begin
  ClientList := New(PCollection, Init(10, 5));
  with ClientList^ do
  begin
    Insert(New(PClient, Init('91-100', 'Anders, Smitty',
                             '(406) 111- 2222')));
    Insert(New(PClient, Init('90-167', 'Smith, Zelda',
                             '(800) 555- 1212')));
    Insert(New(PClient, Init('90-177', 'Smitty, John',
                             '(406) 987- 4321')));
    Insert(New(PClient, Init('90-160', 'Johnson, Agatha',
                             '(302) 139-8913')));
  end;
  PrintAll(ClientList);
  SearchPhone(ClientList, '(406)');
  Dispose(ClientList, Done);
end.
```

Auf die Prozeduren *PrintAll* und *SearchPhone* werden wir noch zurückkommen, aber Sie sehen schon, wie einfach es ist, eine Kollektion zu erstellen. Die erste Anweisung initialisiert die dynamische Instanz *ClientList* von *TCollection*. Diese Kollektion ist anfangs so dimensioniert, daß sie zehn Kundenobjekte aufnehmen kann. Werden mehr als zehn Objekte in der Kollektion gespeichert, wird sie, wann immer notwendig, soweit vergrößert, daß sie

weitere fünf Objekte aufnehmen kann. Die weiteren Anweisungen initialisieren vier Kundenobjekte und speichern sie in der Kollektion. Der Aufruf von *Dispose* am Ende des Ausschnitts führt dazu, daß der Speicherplatz, einschließlich dem, den die Kundenobjekte belegen, freigegeben wird.

Wiederholmethoden

Einträge in eine Kollektion einzufügen und wieder aus ihr zu entfernen, sind nicht die einzigen Operationen, die Sie mit einer Kollektion ausführen müssen. Es wird oft notwendig sein, eine bestimmte Operation auf jeden Eintrag in der Kollektion anzuwenden oder nach dem ersten oder letzten Eintrag zu suchen, der einer bestimmten Bedingung entspricht. Zu diesem Zweck besitzt jede Kollektion drei Wiederholmethoden: *ForEach*, *FirstThat* und *LastThat*. Jede dieser Methoden übernimmt als einzigen Parameter einen Zeiger auf eine Prozedur oder Funktion.

Die Wiederholmethode ForEach

Der Parameter dieser Methode ist ein Zeiger auf eine Prozedur, die ihrerseits als einzigen Parameter einen Zeiger auf einen Eintrag in der Kollektion übernimmt. *ForEach* ruft diese Prozedur für jeden Eintrag in der Kollektion auf. Die Prozedur *PrintAll* ist ein Beispiel für den Einsatz der Wiederholmethode *ForEach*:

```
procedure PrintAll(C: PCollection);  { alle Daten ausgeben }

  procedure PrintClient(P : PClient); far;
  begin
  with P^ do
    Writeln(Account^, '':10-Length(Account^), { Daten des Kunden }
      Name^, '':20-Length(Name^),
      Phone^, '':16-Length(Phone^));
  end;                        { Ende der lokalen Prozedur }

begin
  Writeln;
  Writeln;
  C^.ForEach(@CallPrint);     { f¡r jeden Kunden }
end;
```

Die innerhalb der Prozedur *PrintAll* definierte Prozedur *PrintClient* wird für jeden Eintrag der Kollektion aufgerufen. Sie gibt die Daten dieses Eintrags formatiert aus.

Sie müssen genau darauf achten, welche Art von Prozedur Sie einer Wiederholmethode übergeben. *PrintClient* in unserem Beispiel darf auf keinen Fall

eine Methode des Objekts sein. Sie hat folgenden Bedingungen zu entsprechen:

- Sie muß eine lokal, also innerhalb der Routine, in der sie aufgerufen wird, definierte Prozedur sein.
- Sie muß als **far**-Prozedur deklariert werden, entweder mittels der **far**- oder mittels der Compiler-Direktive **$F+**.
- Sie darf nur einen einzigen Parameter übernehmen, einen Zeiger auf einen Eintrag in der Kollektion.

Die Wiederholmethoden FirstThat und LastThat

Eine bestimmte Operation auf jedes in der Kollektion gespeicherte Element anwenden zu können, ist sicher ganz nützlich. Oft aber wollen Sie auch einen Eintrag finden, der einer bestimmten Bedingung entspricht. Genau dies ist die Aufgabe der Wiederholmethoden *FirstThat* und *LastThat*. Beide übernehmen als Argument einen Zeiger auf eine Funktion, die einen boolschen Wert zurückgibt. Sie durchsuchen die Kollektion, beginnend mit dem ersten (*FirstThat*) oder dem letzten Eintrag (*LastThat*) und geben einen Zeiger auf den ersten Eintrag zurück, für den die übergebene Funktion True ermittelt.

Nehmen wir an, Sie suchten in Ihrer Liste einen bestimmten Kunden, an dessen Namen Sie sich nicht mehr genau erinnern können. Sie wissen aber, daß er in München wohnt. Die Vorwahlnummer von München ist 089, und das erlaubt Ihnen, Ihre Kollektion nach einem Eintrag durchsuchen zu lassen, dessen Feld *Phone* die Zeichenfolge »089« enthält. Die folgende Prozedur durchsucht die Kollektion und verwendet bei der Suche die Wiederholmethode *FirstThat*:

```
procedure SearchPhone(C: PCollection; PhoneToFind: String);

  function PhoneMatch(Client: PClient): Boolean; far;
  begin
    PhoneMatch := Pos(PhoneToFind, Client^.Phone^) <> 0;
  end;

var
  FoundClient: PClient;

begin { SearchPhone }
  Writeln;
  FoundClient := C^.FirstThat(@PhoneMatch);
  if FoundClient = nil then
    Writeln('No client met the search requirement')
  else
  begin
    Writeln('Found client:');
    FoundClient^.Print;
```

```
  end;
end;
```

Wiederum ist die Funktion, die einen Eintrag daraufhin zu überprüfen hat, ob er der Bedingung entspricht, innerhalb der Prozedur definiert, in der auch die Wiederholmethode aufgerufen wird. Auch diese Funktion muß den Bedingungen gehorchen, die oben schon für die *ForEach* zu übergebende Prozedur aufgezählt wurden. Für die Wiederholmethoden *FirstThat* und *LastThat* ist allerdings eine Funktion zu definieren, die einen boolschen Wert zurückgibt. Wird in der Kollektion kein Eintrag gefunden, für den diese Funktion True ermittelt, geben beide Wiederholmethoden einen **nil**-Zeiger zurück.

Sortierte Kollektionen

Einträge werden in einer Kollektion in der Reihenfolge angeordnet, in der sie der Kollektion zur Speicherung übergeben wurden. Dagegen ermöglicht Ihnen das Objekt TSortedCollection, die Einträge in einer von Ihnen festgelegten Reihenfolge in der Kollektion zu speichern.

Dieser Nachkomme von *TCollection* sortiert die ihm übergebenen Objekte, überprüft automatisch die Kollektion, wenn ein neues Element einzufügen ist, und weist doppelte Einträge zurück. Allerdings können Sie die Reaktion des Objekts auf doppelte Einträge ändern. Das Objekt besitzt ein boolsches Feld *Duplicates*, das Sie auf True setzen können, um die Aufnahme von doppelten Einträgen zuzulassen.

TSortedCollection ist ein abstrakter Objekttyp. Um ihn benutzen zu können, müssen Sie zunächst den Typ der Daten definieren, die Sie in der Kollektion speichern wollen. Darauf legen Sie fest, in welcher Reihenfolge diese Daten zu sortieren sind, leiten von *TSortedCollection* einen neuen Objekttyp ab und schreiben die für diese Sortierung zuständigen Methoden *KeyOf* und *Compare*. Den neuen Objekttyp nennen wir im folgenden Beispiel *TClientCollection*.

Dieses Objekt kann bereits Einträge in die Kollektion aufnehmen und wieder aus ihr entfernen. Die Eigenschaften hat es von seinem Vorfahren *TSortedCollection* geerbt. Sie müssen diesem Objekt noch beibringen, welches Feld es als Sortierschlüssel verwenden soll und wie es zwei solcher Schlüssel miteinander vergleichen kann. Dazu müssen die Methoden *KeyOf* und *Compare* überschrieben werden:

```
type
  PClientCollection = ^TClientCollection;
  TClientCollection = object(TSortedCollection)
    function KeyOf(Item: Pointer): Pointer; virtual;
    function Compare(Key1, Key2: Pointer): Integer; virtual;
  end;

function PClientCollection.KeyOf(Item: Pointer): Pointer; virtual;
```

```
begin
  KeyOf := PClient(Item)^.Name;
end;

function PClientCollection.Compare(Key1, Key2: Pointer): Integer;
virtual;
begin
  if PString(Key1)^ = PString(Key2)^ then
    Compare := 0       { 0 zur¡ckgeben, wenn Schl¡ssel gleich }
  else if PString(Key1)^ < PString(Key2)^ then
    Compare := -1      { -1 zur¡ckgeben, wenn Key1 vor Key2 kommt }
  else
    Compare := 1;      { andernfalls kommt Key2 vor Key1 }
end;
```

Die Methode *KeyOf* ermittelt den Sortierschlüssel eines Objekts, in diesem Fall das Feld *Name* des Objekts *TClient*. Der Methode *Compare* werden zwei Zeiger auf solche Schlüssel übergeben, und sie legt fest, wer in der Sortierordnung vor dem anderen liegt. Die Methode gibt entweder -1, 0 oder 1 zurück, je nachdem, ob *Key1* kleiner als, gleich oder größer als *Key2* ist. In diesem Fall werden einfach die beiden Strings miteinander verglichen.

Beachten Sie, daß die Methode *Compare* den Typ der übergebenen Zeiger explizit umwandelt. Dieser Zeiger sind vom Typ Pointer; um sie zu dereferenzieren, müssen sie in den Typ *PString* (ein Zeiger auf einen String) umgewandelt werden.

Das ist alles, was Sie tun müssen, um Ihre Kundenliste zu sortieren. Wenn Sie nun noch *ClientList* als Variable des Typs *PClientCollection* anstatt *PCollection* deklarieren, können Sie die Liste Ihrer Kunden in sortierter Reihenfolge ausgeben lassen. Es genügt, wenn Sie die **var**-Deklaration und den Aufruf von *New* ändern:

```
var
  ClientList: PClientCollection;
...
begin
  ClientList := New(PClientCollection, Init(50, 10));
  ...
end;
```

Strings in einer Kollektion speichern

Turbo Vision stellt eine spezielle Kollektion bereit, die Strings in alphabetischer Ordnung sortiert abspeichert, das Objekt *TStringCollection*. Dieses Objekt nimmt also keine Objekte, sondern nur Zeiger auf Strings auf. Da es ein Nachkomme von *TSortedCollection* ist, werden gleiche Strings nur dann aufgenommen, wenn das Feld *Duplicates* auf True gesetzt wird.

Die Verwendung dieses Objekttyps ist einfach. Sie deklarieren einen Zeiger auf eine Instanz des Objekts und übergeben ihm bei der Initialisierung die Anfangsgröße der Kollektion und einen Wert, der die Schrittweite bestimmt, in der die Kapazität der Kollektion, wenn notwendig, zu erweitern ist:

```
var
  WordList: PStringCollection;
  WordRead String;
  ...
begin
  WordList := New(PStringCollection, Init(10, 5);
  ...
```

WordList nimmt zunächst nur zehn Strings auf, erweitert aber, wenn notwendig, sein Fassungsvermögen um jeweils fünf Strings. Nun können der Kollektion Strings zur Speicherung übergeben. Im folgenden Beispiel sind das einzelne Wörter, die aus einer Textdatei eingelesen werden:

```
repeat
  ...
  if WordRead <> '' then
    WordList^.Insert(NewStr(WordRead));
  ...
until WordRead = '';
...
Dispose(WordList, Done);
```

Die hier benutzte Funktion *NewStr* reserviert den zur Speicherung des Strings notwendigen Speicher, kopiert den String in diesen Bereich und gibt einen Zeiger darauf zurück. Dieser Zeiger wird in der Kollektion eingetragen. Wenn Sie die Kollektion wieder entfernen wollen, müssen Sie sich um die Freigabe des von den Einträgen in der Kollektion belegten Speichers nicht kümmern. Der Destruktor *Done* der Kollektion gibt den von jedem Eintrag belegten Speicher und dann auch den Speicher frei, den das Objekt belegt.

Kollektionen speichern polymorphe Objekte

Was wir bisher in einer Kollektion gespeichert haben, waren Objekte oder auch Daten gleichen Typs. Unsere Kundenliste speicherte Objekte des Typs *TClient*, eine *TStringCollection* kann nur Strings aufnehmen. Aber darin erschöpfen sich die Fähigkeiten einer Kollektion keinewegs.

Der wirkliche Vorteil, den eine Kollektion bietet, besteht darin, daß sie in der Lage ist, Objekte unterschiedlichsten Typs aufzunehmen. Voraussetzung ist lediglich, daß diese Objekte direkt oder indirekt von *TObject* abstammen. Ist diese Voraussetzung erfüllt, dann können Sie in beliebiger Reihenfolge in einer Kollektion abgespeichert werden.

Das Programm TVGUID20.PAS nutzt diese Fähigkeit und speichert drei unterschiedliche graphische Objekte in einer Kollektion. Die Wiederholmethode *ForEach* wird dann dazu benutzt, jedes gespeicherte Objekt auszugeben.

Dieses Demoprogramm bindet die Unit *Graph* ein und benutzt auch die BGI-Treiber. Daher muß sich die Datei GRAPH.TPU entweder im aktuellen Verzeichnis befinden oder in einem der Verzeichnisse, die Sie über *Options/Directories/Unit directory* angemeldet haben, wenn Sie das Programm compilieren. Wechseln Sie, bevor Sie das Programm ausführen lassen, in das Verzeichnis, das die BGI-Treiber enthält, oder ändern Sie den Aufruf von *InitGraph*, indem Sie den Zugriffspfad (zum Beispiel C:\TP\BGI) zu diesem Verzeichnis angeben.

In TVGUID20.PAS wird zunächst ein abstrakter Objekttyp als gemeinsamer Vorfahr aller weiteren graphischen Objekte definiert:

```
type
  PGraphObject = ^TGraphObject;
  TGraphObject = object(TObject)
    X,Y: Integer;
    constructor Init;
    procedure Draw; virtual;
  end;
```

Jeder Nachkomme dieses Objekttyps verfügt bereits, kraft seiner Abstammung, über eine Initialisierungsmethode und eine Methode, die es ihm erlaubt, sich auf dem Bildschirm auszugeben. Nun leiten wir die Objekte Punkt, Kreis und Rechteck von diesem Objekt ab. Den Besonderheiten dieser Objekte entsprechend, müssen dabei die geerbten Methoden eventuell überschrieben werden:

```
  PGraphPoint = ^TGraphPoint;
  TGraphPoint = object(TGraphObject)
    procedure Draw; virtual;
  end;

  PGraphCircle = ^TGraphCircle;
  TGraphCircle = object(TGraphObject)
    Radius: Integer;
    constructor Init;
    procedure Draw; virtual;
  end;

  PGraphRect = ^TGraphRect;
  TGraphRect = object(TGraphObject)
    Width, Height: Integer;
    constructor Init;
    procedure Draw; virtual;
  end;
```

Diese drei Objekttypen erben von ihrem Vorfahren die Felder *X* und *Y*, die eine Position auf dem Bildschirm bezeichnen. Aber die Objekte haben eine

unterschiedliche Größe, die bei *TGraphCircle* durch den Wert des Feldes *Radius*, bei *TGraphRect* durch die Felder *Width* und *Height* bestimmt wird. Instanzen dieser Objekte speichern wir nun in bunter Folge in einer Kollektion ab:

```
procedure MakeCollection(var List: PCollection);
var
  I: Integer;
  P: PGraphObject;
begin
  List := New(PCollection, Init(10, 5));

  for I := 1 to 20 do
  begin
    case I mod 3 of                        { Erstellen }
      0: P := New(PGraphPoint, Init);
      1: P := New(PGraphCircle, Init);
      2: P := New(PGraphRect, Init);
    end;
    List^.Insert(P);                { In der Kollektion speichern }
  end;
end;
```

Wie Sie sehen, speichern wir 20 Objekte in der Kollektion. Eines haben Sie alle gemeinsam, sie sind letztendlich vom Typ *TGraphObject*. Aber Sie wissen nicht, ob ein gespeichertes Objekt nun ein Punkt, ein Kreis oder ein Rechteck ist. Aber das ist auch nicht notwendig, denn jedes Objekt ist in der Lage, sich selbst mittels seiner *Draw*-Methode auf dem Bildschirm darzustellen. Wir benutzen die Wiederholmethode *ForEach*, um die Objekte auszugeben:

```
procedure DrawAll(C: PCollection);

  procedure CallDraw(P : PGraphObject); far;
  begin
    P^.Draw;                          { Draw-Methode aufrufen }
  end;

begin { DrawAll }
  C^.ForEach(@CallDraw);             { Jedes Objekt ausgeben }
end;

var
  GraphicsList: PCollection;
begin
...
  DrawAll(GraphicsList);
...
end.
```

Speicherverwaltung und Kollektionen

Eine *TCollection* kann von ihrer Anfangsgröße, die über *Init* gesetzt wird, zu einer Maximalgröße von 16380 Elementen anwachsen. Diese Maximalgröße speichert Turbo Vision in der Variablen *MaxCollectionSize*. Da nur ein Zeiger abgespeichert wird, verbraucht jedes Element, das Sie einer Kollektion hinzufügen, nur vier Byte Speicher.

Wenn nicht mehr genügend Speicher vorhanden ist, um ein weiteres Element aufzunehmen, wird *TCollection.Error* aufgerufen und ein Laufzeitfehler ausgelöst. Diesem Punkt müssen Sie besondere Beachtung schenken. Wenn es dem Benutzer des Programms überlassen ist, Elemente einer Kollektion hinzuzufügen, dann ist es nicht so leicht vorherzusehen, ob und wann der Heap erschöpft sein wird. Sie müssen also spezielle Maßnahmen treffen, damit der Benutzer Ihrer Programme von einem Laufzeitfehler verschont bleibt, entweder indem Sie den noch verbliebenen Heap vor der Aufnahme eines weiteren Elementes in eine Kollektion überprüfen, oder indem Sie eine Routine schreiben, die Laufzeitfehler abfängt.

9. Streams

Objektorientiert Programmieren heißt, Daten und die Routinen, die mit diesen Daten umgehen, nicht mehr als voneinander getrennte Bereiche zu betrachten, sondern sie als Einheit zu sehen, als ein Objekt. Und auch dieses Objekt ist keine isolierte Größe. Kraft seiner Abstammung ist es eingebunden in eine hierarchische Struktur. Und nun wollen wir etwas eigentlich ganz Einfaches tun: Daten in einer Datei abspeichern.

Das war in jenen Tagen recht einfach, als die Daten, die ein Programm definierte, noch sauber getrennt waren von den Routinen, die diese Daten manipulierten. Doch die Daten, mit denen Turbo Vision arbeitet, sind eingebunden in Objekte. Natürlich können Sie die Objekte zerlegen, die Daten herausnehmen und abspeichern. Aber das wäre ein Schritt in die falsche Richtung. Objektorientierte Programmierung hat es möglich gemacht, Daten und Methoden als eine Einheit zu betrachten, und diese Einheit zerstören heißt, in einen Zustand zurückkehren, den wir gerade überwunden haben.

Wir haben ein Problem und wir kennen dessen Ursache. Sie liegt im Konzept der objektorientierten Programmierung selbst. Können wir dieses Problem nicht auch mittels dieses Konzepts lösen? Turbo Vision hat eine Lösung bereit: *Stream*-Objekte.

Stellen Sie sich einen Stream einfach als eine Gruppe von Objekten vor, die sich auf dem Weg irgendwohin befinden. Das Ziel des Weges kann eine Datei sein, der EMS-Speicher, die serielle Schnittstelle oder irgendein anderes Ein-/Ausgabegerät. Streams erledigen die Ein- und Ausgabe von Objekten, nicht von Daten irgendwelcher Art. Natürlich müssen Sie die Datenfelder Ihrer Objekte irgendwie für einen Stream bereitstellen. Aber diese Vorkehrungen treffen Sie innerhalb Ihrer Objekte selbst.

Das Problem: Objekte speichern und laden

Als Pascal-Programmierer sind Sie daran gewöhnt, daß Sie zuerst den Typ der Daten definieren müssen, bevor Sie diese Daten in eine Datei schreiben oder aus einer Datei lesen können. Auch die Datei selbst muß typisiert sein, und dieser Typ muß feststehen, bevor das Programm compiliert werden kann.

Turbo Pascal erlaubt Ihnen auch, mit untypisierten Dateien zu arbeiten,

aus denen Sie über *BlockRead* Daten lesen und in die Sie mittels *BlockWrite* Daten schreiben können. Sie gewinnen zwar an Flexibilität, wenn Sie solche untypisierten Dateien verwenden, aber das Fehlen jeder Typüberprüfung bürdet Ihnen auch zusätzliche Verantwortung auf.

Bezogen auf Objekte haben wir das Problem, daß Turbo Pascal uns nicht erlaubt, einen Dateityp zu definieren, der auf einem Objekttyp basiert. Und da Objekte auch virtuelle Methoden besitzen können, deren Adressen erst zur Laufzeit des Programms ermittelt werden, ist es ziemlich nutzlos, die in der VMT, der virtuellen Methodentabelle, enthaltenen Daten abzuspeichern. Solche Daten gar in ein Programm einlesen zu wollen, ist noch sinnloser.

Natürlich können Sie auch dieses Problem umgehen, wiederum indem Sie die Objekte zerlegen. Kopieren Sie die Datenfelder der Objekte und schreiben Sie deren Inhalt in irgendeine Datei. Das ist mit aller Zurückhaltung gesagt eine nicht sehr elegante Lösung des Problems und macht die Initialisierung Ihrer Objekte eher komplizierter.

Die Antwort: Streams

Mit Turbo Vision überwinden Sie all diese Schwierigkeiten und verschaffen sich darüber hinaus auch noch einige weitere Vorteile. Ein Stream ist ein einfaches und doch elegantes Mittel, Daten außerhalb Ihres Programmes zu speichern.

Ein Stream vereinigt in sich die Vorteile sowohl typisierter als auch untypisierter Dateien. Der Typ dessen, was Sie abspeichern wollen, wird überprüft, muß aber zum Zeitpunkt der Compilierung noch nicht festgelegt sein. Denn ein Stream weiß, daß er es mit Objekten zu tun hat, und solange dieses Objekt ein Nachkomme von *TObject* ist, kann ein Stream auch damit umgehen.

Alles was Sie tun müssen, ist, dem Stream mitzuteilen, mit welchen Objekttypen er arbeiten muß. Dadurch kann der Stream die Daten der richtigen VMT zuordnen. Danach können Sie ohne große Mühe Objekte dem Stream übergeben und wieder aus dem Stream lesen.

Aber wie kann ein Stream so ganz unterschiedliche Objekte wie ein *TDeskTop* oder ein *TDialog* schreiben und lesen, ohne daß ihm zum Zeitpunkt der Compilierung mitgeteilt wird, welche Objekte er verarbeiten muß? Das ist wirklich sehr verschieden von dem, wie Pascal normalerweise die Ein- und Ausgabe handhabt.

Die Antwort darauf: *Registrierung*. Jedem Objekttyp in Turbo Vision, und auch den Objekttypen, die Sie von schon vorhandenen ableiten, wird eine eindeutige Registrierungsnummer zugewiesen. Diese Nummer wird vor den Daten des Objekts in den Stream geschrieben, und wenn Turbo Vision dieses Objekt wieder einlesen muß, wird zunächst die Nummer gelesen, wodurch Turbo Vision erfährt, wieviele Daten insgesamt zu lesen und welcher VMT diese Daten zuzuordnen sind.

Wie Streams zu benutzen sind

Einen Stream können Sie durchaus mit einer üblichen sequentiellen Pascal-Datei vergleichen. Auch in diese schreiben Sie Daten und lesen sie dann wieder ein. Eine solche Datei ist eine einfache Ein-/Ausgeeinheit, in die Sie Daten eines bestimmten Typs speichern und dann wieder laden können. Ein Stream ist unter diesem Aspekt eine Ein-/Ausgeeinheit, in die Sie *polymorphe* Objekte schreiben und wieder aus ihr lesen können.

Pascal-Dateien erlauben den wahlfreien Zugriff auf die gespeicherten Daten. Sie setzen den Dateizeiger auf eine bestimmte Position in der Datei, lesen an dieser Position Daten ein, versetzen wiederum den Zeiger usw. Diese Operationen sind auch mit einem Stream möglich. Wir gehen im Abschnitt »Streams mit wahlfreiem Zugriff« näher darauf ein.

Wie eine Datei auch, müssen Sie einen Stream öffnen, bevor Sie in diesen Stream etwas schreiben oder aus ihm einlesen können. Danach muß dieser Stream wieder geschlossen werden. Das notwendige Vorgehen beschreiben wir in den nächsten Abschnitten.

Den Stream öffnen

Bevor Sie mit einem Stream arbeiten können, müssen Sie ihn zunächst initialisieren. Die genaue Syntax des *Init*-Konstruktors, den Sie zu diesem Zweck aufrufen, hängt von dem Streamtyp ab, den Sie benutzen. Wenn Sie zum Beispiel einen *TDosStream* öffnen, müssen Sie den Namen der DOS-Datei angeben und zusätzlich den Zugriffsmodus bestimmen (Datei nur zum Lesen, nur zum Schreiben oder für beide Operationen öffnen).

Haben Sie den Stream einmal initialisiert, können Sie mit ihm arbeiten. Mehr brauchen Sie nicht zu tun.

TStream ist ein abstrakter Objekttyp, von dem Sie sinnvollerweise nie eine Instanz definieren. Er ist aber der gemeinsame Vorfahre aller nützlichen Stream-Objekte, darin eingeschlossen *TDosStream*, das Ein- und Ausgabe über eine Datei abwickelt, *TBufStream*, das dieselben Operationen über einen Puffer erledigt, wodurch die Ein- und Ausgabe vieler kleiner Datenmengen beschleunigt wird, und *TEmsStream*, das auf EMS-Speicher zugreift und dadurch gerade für Ressourcen, auf die schnell zugegriffen werden muß, interessant ist.

Turbo Vision stellt auch ein Stream-Objekt zur Verfügung, dessen Einträge indiziert werden. Über einen Positionszeiger können Sie dann auf bestimmte Einträge zugreifen. Dieses Objekt realisiert damit einen Stream mit wahlfreiem Zugriff.

In den Stream schreiben, aus dem Stream lesen

Der Basistyp TStream definiert drei Methoden, auf die wir näher eingehen müssen: *Get*, *Put* und *Error*. *Get* und *Put* entsprechen im wesentlichen den Prozeduren *Read* und *Write*, die Sie für normale Ein- und Ausgaben in und aus Dateien benutzen. Die Methode *Error* wird immer dann aufgerufen, wenn im Stream ein Fehler aufgetreten ist.

Das Objekt abschicken

Betrachten wir zunächst die Prozedur *Put*. Die folgende Anweisung zeigt die Syntax des Aufrufs dieser Methode:

```
SomeStream.Put(PSomeObject);
```

wobei *SomeStream* irgendein initialisiertes und von *TStream* abgeleitetes Objekt bezeichnet und *PSomeObject* einen Zeiger auf ein von *TObjekt* abgeleitetes Objekt darstellt, dessen Typ dem Stream über die Registrierung bekannt ist. Mehr müssen Sie nicht angeben. Der Stream kann aus der VMT von *PSomeObjekt* selbst den Typ des Objekts ermitteln, weiß also, welche Identifikationsnummer und wieviele Daten zu schreiben sind.

Möglicherweise veranlaßt ein Aufruf von *Put*, daß mehr als ein Objekt in den Stream geschrieben wird. Ist nämlich dieses Objekt eine Gruppe, wird nicht nur es selbst, sondern auch alle Subviews, die zu dieser Gruppe gehören, in den Stream gespeichert. Diese Subviews wiederum können natürlich auch Gruppen sein, in welchem Fall auch deren Subviews geschrieben werden. Sie können den aktuellen Zustand Ihres Programms sichern, wenn Sie die Arbeitsfläche, also das Objekt *TDeskTop* in einen Stream schreiben. Starten Sie dann Ihr Programm neu und laden die Arbeitsfläche wieder. Sie finden denselben Zustand Ihres Programms wieder, in dem es sich befand, als Sie die Arbeitsfläche haben abspeichern lassen.

Das Objekt zurückholen

So einfach, wie sich Objekte abspeichern lassen, so einfach können Sie sie wieder einlesen. Sie müssen nur wie in der folgenden Anweisung die Funktion *Get* aufrufen:

```
PSomeObject := SomeStream.Get;
```

wobei wieder gilt, daß *SomeStream* ein initialisiertes Stream-Objekt bezeichnet und *PSomeObject* ein Zeiger auf ein Objekt ist. *Get* gibt einfach einen Zeiger auf das zurück, was diese Funktion aus dem Stream gelesen hat. Wieviele Daten gelesen und welcher VMT sie zugeordnet wurden, ist festgelegt durch den Typ des Objekts, der im Stream gefunden wurde, und nicht etwa durch den Typ von *PSomeObject*. Wenn daher das Objekt im Stream nicht dem Typ des durch *PSomeObject* referenzierten Objekts entspricht, werden Sie mit den gelesenen Daten nichts anfangen können.

So wie *Put* komplexe Objekte sichert, liest auch *Get* solche Objekte wieder ein. Wenn also das gelesene Objekt eine Gruppe ist, zu der weitere Subviews gehören, liest *Get* auch diese ein.

Im Falle eines Fehlers

Die Prozedur *Error* legt schließlich fest, was geschieht, wenn im Stream ein Fehler auftritt. So wie sie ist, beschränkt sich diese Methode darauf, einen solchen Fehler durch entsprechende Belegung der Felder *Status* und *ErrorInfo* zu dokumentieren. Sie müssen diese Methode überschreiben, um im Falle eines Fehlers zum Beispiel ein Dialogfenster zu öffnen oder einen Laufzeitfehler auszulösen.

Den Stream schließen

Wenn Sie den Stream nicht weiter benutzen wollen, rufen Sie die *Done*-Methode auf, so wie Sie bei einer normalen Datei die Prozedur *Close* aufrufen würden. Die Syntax dieser Anweisung entspricht im Kern der, die für jedes Turbo-Vision-Objekt gilt:

```
Dispose(SomeStream, Done);
```

Dadurch wird der Stream geschlossen und der von diesem Objekt belegte Speicher freigegeben.

Objekte für einen Stream vorbereiten

Alle in Turbo Vision definierten Objekte sind bereits für die Arbeit mit einem Stream vorbereitet (im Abschnitt »Standardobjekte« gehen wir genauer darauf ein). Wenn Sie von einem dieser Standardobjekte ein neues Objekt ableiten, müssen Sie dieses Objekt einerseits für einen Stream vorbereiten und andererseits dem Stream die Existenz dieses Objekttyps mitteilen. Beides ist ziemlich einfach.

Die Methoden Store und Load

Wirklich geschrieben und gelesen werden die Objekte von den Methoden *Store* und *Load*. Jedes Objekt, das in einen Stream geschrieben und aus einem Stream gelesen werden soll, muß über diese beiden Methoden verfügen. Keine dieser Methoden jedoch rufen Sie direkt auf. Sie werden vielmehr von den Methoden *Put* und *Get* des Streams aufgerufen. Daher müssen Sie nur dafür sorgen, daß Ihr Objekt in der Lage ist, sich selbst im Stream zu speichern, wenn es dazu aufgefordert wird.

Da wir es mit Objekten zu tun haben, ist dieses Vorhaben leicht zu realisieren, denn die meiste Arbeit erledigen die Vorfahren des Objekts. Sie müssen nur dafür sorgen, daß die Datenfelder, die Ihr Objekt neu enthält, in

den Stream geschrieben werden. Den Rest übernehmen die *Store*- und *Load*-Methoden der Vorfahren.

Nehmen wir an, Sie leiten von *TWindow* ein neues Objekt ab:

```
type
  TMagritte = object(TWindow)
    Painted : Boolean;
    constructor Load(var S: TStream);
    procedure Draw;
    procedure Store(var S: TStream);
  end;
```

Das einzig neue Feld, das wir diesem Objekt hinzugefügt haben, ist das boolsche Feld *Painted*. Daher können wir über die geerbte *Load*-Methode ein ganz normales *TWindow*-Objekt einlesen lassen, und müssen nur noch dafür sorgen, daß unsere Methode ein weiteres Byte für dieses boolsche Feld liest. Das Gleiche gilt für die Methode *Store*. Wir lassen ein *TWindow*-Objekt speichern und schreiben dann ein weiteres Byte in den Stream:

```
constructor TMagritte.Load(var S: TStream);
begin
  TWindow.Load(S);              { liest den ererbten Typ }
  S.Read(Painted, SizeOf(Boolean));      { das neue Feld }
end;

procedure TMagritte.Store(var S: TStream);
begin
  TWindow.Store;             { schreibt den ererbten Typ }
  S.Write(Painted, SizeOf(Boolean));     { das neue Feld }
end;
```

Hier ist eine **Warnung** notwendig. Es liegt allein in Ihrer Verantwortung sicherzustellen, daß genauso viele Daten in der gleichen Reihenfolge gelesen werden, wie sie geschrieben wurden. Der Compiler hat keine Möglichkeit, das zu überprüfen. Wenn Sie also nicht sorgfältig vorgehen, handeln Sie sich einige Probleme ein. Wenn Sie in einem Objekt irgendwelche Datenfelder ändern, müssen Sie diese Änderungen auch in den Methoden *Store* und *Load* berücksichtigen.

Objekte registrieren

Nun müssen Sie dem Stream nur noch mitteilen, daß ein weiterer Objekttyp zur Bearbeitung vorliegt. Zu dieser Registrierung sind zwei Schritte erforderlich. Sie definieren einen Datenrecord des Typs *TStreamRec* und übergeben diesen Record der global verfügbaren Prozedur *RegisterType*. Der Record, mittels dessen ein Objekt registriert wird, ist folgendermaßen definiert:

```
PStreamRec = ^TStreamRec;
TStreamRec = record
```

```
    ObjType: Word;
    VmtLink: Word;
    Load: Pointer;
    Store: Pointer;
    Next: Word;
end;
```

In Turbo Vision tragen alle Registrierungsrecords den gleichen Namen wie das Objekt, das sie anmelden, allerdings ist der erste Buchstabe ein »*R*« statt eines »*T*«. Der Registrierungsrecord für *TDeskTop* heißt also *RDeskTop*. Folgen wir dieser Konvention und nennen den Record für unser neues Objekt *RMagritte*. In Turbo Vision haben übrigens abstrakte Objekttypen wie *TObject* oder *TView* keinen solchen Record, da von solchen Objekten nie eine Instanz in einem Stream zu speichern ist.

Ein Objekt identifiziert sich

Das einzige Feld in einem Registrierungsrecord, um das Sie sich wirklich kümmern müssen, ist das Feld *ObjType*, das die Registrierungsnummer aufnimmt. Jeder Objekttyp muß eine eindeutige Registrierungsnummer besitzen. Turbo Vision reserviert die Nummern 0 bis 99 für die Standardobjekte, so daß Sie die Nummern 100 bis 65535 für Ihre Objekte verwenden können.

Es liegt in Ihrer Verantwortung, den Objekten, die Sie definieren, entsprechende Nummern zuzuordnen, und den Benutzern Ihrer Units mitzuteilen, welchem Objekt welche Nummer zugeteilt wurde. Wie bei Befehlskonstanten auch, können Sie die Registrierungsnummer für Ihre Objekte innerhalb der gegebenen Grenzen frei bestimmen, solange sie nur eindeutig sind.

Die Verbindung mit dem Objekt

wird über das Feld *VmtLink* hergestellt, denn es verweist auf die virtuelle Methodentabelle (VMT) Ihres Objekts. Sie belegen es einfach mit dem Offset auf Ihren Objekttyp:

```
RSomeObject.VmtLink := Ofs(TypeOf(TSomeObject)^);
```

Die Felder *Load* und *Store* speichern die Adressen der gleichnamigen Methoden Ihres Objekts:

```
RSomeObject.Load := @TSomeObject.Load;
RSomeObject.Store := @TSomeObject.Store;
```

Das letzte Feld des Records wird von *RegisterType* selbst belegt, Sie brauchen sich also darum nicht zu kümmern. Es dient dazu, den Aufbau einer verketteten Liste solcher Registrierungsrecords zu erleichtern.

Registrieren Sie hier

Wenn Sie einen solchen Registrierungsrecord für Ihr Objekt »ausgefüllt« haben, rufen Sie die Prozedur *RegisterTypes* auf und übergeben ihr den Record. Die folgenden Anweisungen registrieren das neue Objekt *TMagritte*:

```
const
  RMagritte: TStreamRec = (
    ObjType: 100;
    VmtLink: Ofs(TypeOf(TMagritte)^);
    Load: @TMagritte.Load;
    Store: @TMagritte.Store;
  );

  RegisterType(RMagritte);
```

Nun ist das Objekt registriert und Sie können beliebige Instanzen dieses Objekts in jedem in Turbo Vision definierten Stream abspeichern und wieder einlesen.

Standardobjekte registrieren

Wie erwähnt sind die Standardobjekte in Turbo Vision schon für die Arbeit mit einem Stream vorbereitet, das heißt, Turbo Vision definiert für jedes dieser Objekte einen Registrierungsrecord. Die Registrierung selbst müssen jedoch Sie veranlassen. Um Ihnen dies zu erleichtern, definiert jede Unit eine Prozedur *RegisterXXXX*, wobei *XXXX* für den Namen der jeweiligen Unit steht. Die folgende Aufstellung listet für jede dieser Prozeduren die Objekte, die sie registriert. Hinter den Namen der Objekte ist in Klammern die Identifikationsnummer diese Objekttyps angegeben.

- Die Prozedur *RegisterApp* (definiert in der Unit *App*) registriert die Objekte *TBackGround* (30) und *TDeskTop* (31).

- Die Prozedur *SelRegisterColorSel* (definiert in der Unit *ColorSel*) registriert die Objekte *TColorSelector* (21), *TColorDisplay* (22), *TColorGroupList* (23), *TColorItemList* (24), und *TColorDialog* (25).

- Die Prozedur *RegisterDialogs* (definiert in der Unit *Dialogs*) registriert die Objekte *TDialog* (10), *TInputLine* (11), *TButton* (12), *TCluster* (13), *TRadioButtons* (14), *TCheckBoxes* (15), *TListBox* (16), *TStaticText* (17), *TLabel* (18), *THistory* (19) und *TParamText* (20).

- Die Prozedur *RegisterMenus* (definiert in der Unit *Menus*) registriert die Objekte *TMenuBar* (40), *TMenuBox* (41) und *TStatusLine* (42).

- Die Prozedur *RegisterObjects* (definiert in der Unit *Objects*) registriert die Objekte *TCollection* (50) und *TStringCollection* (51).

■ Die Prozedur *RegisterViews* (definiert in der Unit *Views*) registriert die Objekte *TView* (1), *TFrame* (2), *TScrollBar* (3), *TScroller* (4), *TListViewer* (5), *TGroup* (6) und *TWindow* (7).

Mithilfe dieser Prozeduren können Sie gezielt jene Standardobjekte registrieren lassen, von denen Ihr Programm Gebrauch macht. Beachten Sie aber, daß keine der Registrierungsprozeduren die Objekte *TStringList* und *TStrListMaker* registriert, denn beiden wurde die gleiche (52) Registrierungsnummer zugewiesen. Wenn also Ihr Programm eines dieser beiden Objekte verwendet, müssen Sie es selbst registrieren.

Wie ein Stream funktioniert

Wir wissen jetzt, wie ein Objekt für einen Stream vorbereitet wird, und welche Aufgaben ein Stream übernimmt. Betrachten wir jetzt ein wenig genauer, *wie* er diese Aufgaben erledigt. Dies ist auch ein gutes Beispiel dafür, wie Objekte zusammenarbeiten, und wie sie die Arbeit untereinander aufteilen.

Ein Objekt speichern

Wenn Sie über die *Put*-Methode eines Streams diesem ein Objekt übergeben, holt sich der Stream zunächst einmal die Adresse der VMT des zu speichernden Objekts. Dann durchsucht er die Liste der registrierten Objekte nach einer Übereinstimmung. Wenn gefunden, wird die Identifikationsnummer des Objekts abgespeichert. Darauf ruft der Stream die *Store*-Methode des Objekts auf. Diese Methode schließlich schreibt die Daten des Objekts, wobei sie sich der Prozedur *Write* des Streams bedient.

Ihr Objekt braucht über den Stream nichts zu wissen. Mag dieser nun eine DOS-Datei oder der EMS-Speicher sein, das Objekt begnügt sich damit, den Stream aufzufordern, es in den Stream zu speichern, und dieser erledigt den Rest.

Ein Objekt lesen

Lassen Sie ein Objekt von einem Stream lesen, lädt *Get* zunächst die Identifikationsnummer und durchsucht dann die Liste der registrierten Objekte nach einer Übereinstimmung. Wenn gefunden, ermittelt *Get* aus dem Record die Position der VMT und die Adresse der *Load*-Methode. Diese Methode wird dann aufgerufen, und sie liest die richtige Menge Daten aus dem Stream ein.

Im Prinzip sagen Sie dem Stream also, er solle das nächste Objekt, das im Stream enthalten ist, einlesen und einen Zeiger darauf zurückzuliefern. Ihr Objekt kümmert sich nicht darum, aus welchem Stream es eingelesen wird. Der Stream sorgt dafür, daß die passende Menge Daten eingelesen wird, verläßt sich dabei aber auf die *Load*-Methode des Objekts, die wiederum das eigentliche Lesen der Daten von der Methode *Read* des Streams erledigen läßt.

All dies vollzieht sich, ohne daß Sie weiter eingreifen müßten, zeigt aber, wie wichtig es ist, einen neuen Objekttyp zu registrieren, bevor Sie Instanzen dieses Objekts in einen Stream schreiben oder aus einem Stream lesen lassen.

Objekte, die keine sind

Noch eine Bemerkung zu »Objekten« mit der Registrierungsnummer 0. Sie können auch ein **nil**-Objekt in einen Stream speichern. In einem solchem Fall verwendet der Stream die Nummer 0 als Identifikationsnummer für dieses »Objekt«. Wenn der Stream bei einer Leseoperation auf die Registrierungsnummer 0 trifft, gibt er einen **nil**-Zeiger zurück. Daher ist die Nummer 0 reserviert und steht als Identifikationsnummer nicht zur Verfügung.

Kollektionen und Streams: ein Beispiel

In Kapitel 8, »Kollektionen«, haben Sie gesehen, daß eine Kollektion unterschiedliche Objekte aufnehmen kann, eine Eigenschaft, die auch für einen Stream zutrifft. Ein Stream kann sogar eine Kollektion in einer Datei speichern und sie später wieder einlesen. Betrachten wir noch einmal TVGUID20.PAS. Wie erreichen wir, daß die Kollektion in einem Stream gespeichert werden kann?

Dieses Vorhaben ist bemerkenswert einfach zu realisieren. Sie müssen zunächst dem Basis-Objekt *TGraphObject* beibringen, wie es seine Felder einem Stream übergeben muß. Das ist die Aufgabe der Methode *Store*, die Sie für dieses Objekt definieren. In den von *TGraphObject* abgeleiteten Objekten überschreiben Sie die geerbte *Store*-Methode, damit auch die in diesen Objekten neu definierten Felder abgespeichert werden. Das Objekt *TGraphCircle* muß das Feld *Radius*, das Objekt *TGraphRect* die Felder *Width* und *Height* abspeichern.

Definieren Sie dann die notwendigen Registrierungsrecords für die neuen Objekttypen, die in einem Stream zu speichern sind. Beim Start des Programm registrieren Sie diese Objekte. Das ist eigentlich alles. Denn der Rest ist ähnlich der Ein- und Ausgabe über eine normale Datei. Sie deklarieren eine Stream-Variable, initialisieren den Stream, speichern über eine einzige Anweisung die gesamte Kollektion und schließen den Stream.

Store-Methoden einfügen

Hier folgen nun die *Store*-Methoden der Objekte. Nur *PGraphPoint* benötigt keine solche Methode, denn dieses Objekt definiert keine neuen Felder.

```
type
  PGraphObject = ^TGraphObject;
  TGraphObject = object(TObject)
    ...
    procedure Store(var S: TStream); virtual;
  end;

  PGraphCircle = ^TGraphCircle;
  TGraphCircle = object(TGraphObject)
    ...
    procedure Store(var S: TStream); virtual;
  end;

  PGraphRect = ^TGraphRect;
  TGraphRect = object(TGraphObject)
    ...
    procedure Store(var S: TStream); virtual;
  end;
```

Die Definition der Store-Methoden folgt dem gleichen Muster. Sie rufen die *Store*-Methode des Vorfahren auf und schreiben dann über *TStream.Write* die Felder in den Stream, die das Objekt definiert. Diesem Muster folgt nur das Objekt *TGraphObject* nicht. Sein Vorfahre ist *TObject*, und dieses Objekt besitzt keine Felder, die in den Stream zu schreiben wären.

```
procedure TGraphObject.Store(var S: TStream);
begin
  S.Write(X, SizeOf(X));
  S.Write(Y, SizeOf(Y));
end;

procedure TGraphCircle.Store(var S: TStream);
begin
  TGraphObject.Store(S);
  S.Write(Radius, SizeOf(Radius));
end;

procedure TGraphRect.Store(var S: TStream);
begin
  TGraphObject.Store(S);
  S.Write(Width, SizeOf(Width));
  S.Write(Height, SizeOf(Height));
end;
```

Die Methode TStream.Write schreibt eine bestimmte Anzahl Bytes in den Stream. Der erste Parameter dieser Methode ist eine Variable beliebigen Typs,

der zweite muß die Anzahl der zu schreibenden Bytes angeben. Wir haben diese Anzahl immer über die Standardfunktion *SizeOf* ermitteln lassen. Wenn wir uns später entschließen, den Typ eines Feldes zu ändern, sagen wir in einen LongInt statt eines Integer, brauchen wir uns um die Anzahl der zu schreibenden Bytes nicht zu kümmern. *SizeOf* ermittelt automatisch die richtige Anzahl: vier Bytes statt wie vorher zwei.

Die Objekte registrieren

Unsere Objekte müssen wir nun noch dem Stream bekanntmachen. Voraussetzung dazu ist die Definition der entsprechenden Registrierungsrecords. Diese Records benennen wir in der Weise, wie es Turbo Vision auch tut. Wir verwenden den Objektnamen, ersetzen jedoch dessen führendes »*T*« durch »*R*«. Für jeden Objekttyp vergeben wir eine eindeutige Identifikationsnummer, wobei wir beachten, daß Turbo Vision den Bereich 0 bis 99 für eigene Zwecke reserviert:

```
const
  RGraphPoint: TStreamRec = (
    ObjType: 150;
    VmtLink: Ofs(TypeOf(TGraphPoint)^);
    Load: nil;                    { Noch keine Load-Methode }
    Store: @TGraphPoint.Store);

  RGraphCircle: TStreamRec = (
    ObjType: 151;
    VmtLink: Ofs(TypeOf(TGraphCircle)^);
    Load: nil;                    { Noch keine Load-Methode }
    Store: @TGraphCircle.Store);

  RGraphRect: TStreamRec = (
    ObjType: 152;
    VmtLink: Ofs(TypeOf(TGraphRect)^);
    Load: nil;                    { Noch keine Load-Methode }
    Store: @TGraphRect.Store);
```

Für das Objekt *TGraphObject* wurde kein Registrierungsrecord definiert, denn von diesem abstrakten Objekttyp erzeugen wir nie eine Instanz, die im Stream zu speichern wäre. Das Feld *Load* der Records ist auf **nil** gesetzt, da wir bisher keine *Load*-Methoden definiert haben.

Die eigentliche Registrierung übernimmt eine eigene Prozedur, die wir dann am Beginn unseres Programms aufrufen:

```
procedure StreamRegistration;
begin
  RegisterType(RCollection);
  RegisterType(RGraphPoint);
  RegisterType(RGraphCircle);
```

```
  RegisterType(RGraphRect);
end;
```

Beachten Sie, daß Sie auch das Objekt TCollection registrieren müssen. In Ihrer Verantwortung liegt es, alle Objekttypen registrieren zu lassen, die Ihr Programm einbindet und die mit einem Stream zusammenarbeiten sollen. Und das schließt auch diejenigen ein, die Turbo Vision bereitstellt. Für diese Standardobjekte ist zwar schon ein Registrierungsrecord definiert, die Registrierung selbst müssen jedoch Sie veranlassen.

In den Stream schreiben

Was nun noch folgt, entspricht der Ausgabe in eine normale Datei. Sie deklarieren eine Stream-Variable, initialisieren den Stream, speichern über eine einzige Anweisung die gesamte Kollektion und schließen den Stream. Die folgenden Anweisungen sind ein Ausschnitt aus TVGUID21.PAS:

```
var
  GraphicsList: PCollection;
  GraphicsStream: TBufStream;
begin
  StreamRegistration;                  { Objekte registrieren }
  ...
  { Kollection in einem Stream schreiben }
  GraphicsStream.Init('GRAPHICS.STM', stCreate, 1024);
  GraphicsStream.Put(GraphicsList);    { Kollection schreiben }
  GraphicsStream.Done;
  ...
end.
```

Damit wird eine Datei erzeugt und die gesamte Kollektion der Grafikobjekte gespeichert. Das Programm TVGUID22.PAS definiert für diese Objekte auch die passenden *Load*-Methoden, mittels derer die Objekte wieder aus der Datei geladen werden können. Sie sehen: Der Einsatz einer Kollektion in Verbindung mit einem Stream ist einfach, der Gewinn enorm. Denn Sie verfügen damit über die Mittel, eine ganze Gruppe polymorpher Objekte in einer Datei ablegen und wieder aus ihr lesen zu können.

Zeiger in einem Stream speichern?

Ein spezielles Problem stellen solche Objekte dar, die in ihren Datenfeldern Zeiger auf andere Objekte speichern. Natürlich können Sie auch Zeiger in einem Stream speichern und wieder aus dem Stream laden. Sie dürfen dann aber nicht erwarten, daß dieser Zeiger noch das Objekt referenziert, das Sie

über dieses Datenfeld ansprechen wollten. Und wenn Sie mit diesem Zeiger arbeiten, werden Sie bald feststellen, daß sich Ihr Programm sehr seltsam verhält – im besten Fall! Sehr viel wahrscheinlicher ist, daß es einfach abstürzt.

Aber auch für dieses Problem bietet Turbo Vision eine Lösung. Wir müssen dabei aber zwei Fälle unterscheiden.

Zeiger auf Subviews

Betrachten wir einmal ein Dialogfenster. Dieses Objekt ist eine Gruppe, zu der einige Dialogelemente gehören. Manchmal mag es ganz nützlich sein, wenn das Dialog-Objekt ein Feld definiert, das einen Zeiger auf eine dieser Subviews, sagen wir einen Aktionsschalter, speichert. Das bietet die Möglichkeit, sehr viel einfacher auf das referenzierte Objekt zuzugreifen. Nun speichern wir die Gruppe in einem Stream, wobei auch der Zeiger abgespeichert wird, und geben den belegten Speicher wieder frei. Zu einem späteren Zeitpunkt lesen wir die Gruppe wieder ein, einschließlich des Zeigers. Wohin wird er zeigen? Dorthin, wohin er auch vorher zeigte. Auf was zeigt er? Wer weiß. Jedenfalls nicht mehr auf das Objekt Aktionsschalter. Der einzige Schalter, der jetzt noch Sinn macht, ist die Resettaste.

Die Lösung für dieses Problem bieten die Methoden *TGroup. GetSubViewPtr* und *TGroup. PutSubViewPtr*. Statt einfach einen Zeiger auf ein Objekt abzuspeichern, rufen Sie die Methode *PutSubViewPtr* auf, die einen numerischen Wert abspeichert. Welche Bedeutung hat dieser Wert? Jede Gruppe verwaltet eine Liste der Objekte, die zu der Gruppe gehören. Der Wert, den *PutSubViewPtr* abspeichert, ist nichts anderes als die Position des entsprechenden Objekts in dieser Liste. Wenn Sie dann die Gruppe wieder einlesen, rufen Sie *GetSubViewPtr* auf und diese Methode liest den Wert ein, und gibt einen Zeiger auf das Objekt zurück, das an der entsprechenden Position in der Liste steht.

Der folgende Programmauschnitt zeigt Ihnen, wie Sie diese beiden Methoden verwenden:

```
type
  TButtonWindow = object(TWindow)
    Button: PButton;
    constructor Load(var S: TStream);
    procedure Store(var S: Stream);
  end;

constructor TButtonWindow.Load(var S: TStream);
begin
  TWindow.Load(S);
  GetSubViewPtr(S, Button);
end;

procedure TButtonWindow.Store(var S: TStream);
begin
  TWindow.Store(S);
```

```
  PutSubViewPtr(S, Button);
end;
```

Die neue Methode *Store* ruft zunächst die Methode seines Vorfahren auf, die das Fenster und dabei natürlich auch den Aktionsschalter, der zu dieser Gruppe gehört, speichert. Darauf wird über *PutSubViewPtr* eine Referenz auf dieses Objekt in den Stream geschrieben. Diese Referenz läßt die *Load*-Methode über *GetSubViewPtr* wieder einlesen, und diese Methode ermittelt anhand dieser Referenz einen Zeiger auf das Objekt.

Auf Kollegen zeigen

Eine ähnliche Situation entsteht, wenn ein Objekt ein Feld besitzt, das einen Zeiger auf ein *View*-Objekt speichert, das zu der gleichen Gruppe wie das Objekt selbst gehört. Betrachten wir ein *TScroller*-Objekt. Dieses Objekt enthält zwei Zeiger, die auf Rollbalken-Objekte verweisen. Diese Rollbalken gehören jedoch nicht zu dem *TScroller*-Objekt, sondern wie dieses auch zu der Gruppe *TWindow*. Es handelt sich also sozusagen um Kollegen.

Auch hier macht es wenig Sinn, diese Zeiger abzuspeichern. Und auch hier bietet Turbo Vision eine Lösung. Die Methoden *TView.PutPeerViewPtr* und *TView.GetPeerViewPtr* greifen auf die Liste der Subviews zu, die von der Gruppe verwaltet wird, zu der die betroffenen Objekte gehören.

Aber es könnte ja auch der Fall eintreten, daß ein Objekt der Gruppe ein anderes Objekt referenziert, das noch gar nicht aus dem Stream gelesen wurde. Das kann passieren, wenn dieses Objekt in der Subview-Liste weiter hinten steht und damit auch erst später aus dem Stream gelesen wird. Wir hätten es also mit einer Vorwärtsreferenz zu tun, die erst aufgelöst werden kann, wenn alle zur Gruppe gehörenden Objekte geladen sind. Und genau dies erledigt Turbo Vision automatisch. Sie müssen sich trotzdem dieses Problems bewußt sein. Denn wenn Sie etwa in einer *Load*-Methode irgendwelche Operationen auf Objekte mit solchen noch nicht aufgelösten Referenzen anwenden, wird das Ergebnis kaum angenehm sein.

Die Arbeitsfläche speichern und wieder einlesen

Wenn das Objekt, das Sie in einem Stream abspeichern, die Arbeitsfläche selbst ist, wird dieses Objekt auch die Speicherung aller zu ihm gehörenden Subviews veranlassen. Wollen Sie für Ihr Programm vorsehen, daß der Benutzer die Arbeitsfläche speichern kann, müssen Sie sicherstellen, daß jedes *View*-Objekt über geeignete *Store*- und *Load*-Methoden verfügt, und daß jedes Objekt registriert wurde. Das ist notwendig, weil Sie nicht vorhersehen können, welche Objekte zu dem Zeitpunkt auf dem Bildschirm dargestellt sind, in dem der Benutzer beschließt, die Arbeitsfläche zu speichern.

Der folgende Ausschnitt zeigt, wie Sie vorgehen müssen, um ein solch komplexes Objekt wie die Arbeitsfläche zu laden:

```
procedure TMyApp.RestoreDeskTop;
var
  SaveFile: TBufStream;          { ein gepufferter Stream }
  Temp: PDeskTop;
begin
  SaveFile.Init('T.DSK', stOpen, 1024);   { Datei ·ffnen }
  Temp := PDeskTop(SaveFile.Get);  { Arbeitsfl/che lesen }
  SaveFile.Done;                        { Datei schlieøen }
  if Temp <> nil then                 { wenn etwas gelesen wurde.. }
  begin
    Dispose(DeskTop, Done);           { ...l·schen wir die
                                      alte Arbeitsfl/che }
    DeskTop := Temp;                { ...ersetzen sie durch
                                                die neue }
    Insert(DeskTop);              { ...und f¡gen sie in das
                                              Programm ein }
    DeskTop^.DrawView;            { zeigen, was wir haben }
  end;
  if SaveFile.Status <> 0 then ErrorReadingFile;
end;
```

Einen Stream kopieren

TStream definiert auch die Methode *CopyFrom(S, Count)*, die *Count* Bytes aus einem Stream in einen anderen kopiert. Diese Methode können Sie benutzen, wenn Sie fortgesetzt auf einen Datei-Stream zugreifen müssen, aber es aus Geschwindigkeitsgründen vorziehen würden, den EMS-Speicher zu nutzen:

```
NewStream := New(TEmsStream, Init(OldStream^.GetSize));
OldStream^.Seek(0);
NewStream^.CopyFrom(OldStream, OldStream^.GetSize);
```

Streams mit wahlfreiem Zugriff

Bisher haben wir Streams ausschließlich als sequentielle Ein-/Ausgabeeinheiten betrachtet. Sie schreiben ein Objekt nach dem anderen in den Stream und lesen Sie dann in der gleichen Reihenfolge wieder ein. Turbo Vision bietet aber weitaus mehr.

Turbo Vision ermöglicht Ihnen auch, einen Stream als virtuelle Ein-/Aus-

gabeeinheit mit wahlfreiem Zugriff zu behandeln. Denn über *Get* und *Put* hinaus bietet ein Stream-Objekt ähnliche Möglichkeiten, wie die, die bei einer normalen Datei über *Seek*, *FilePos*, *FileSize* und *Truncate* zur Verfügung stehen.

Die Prozedur *Seek* eines Stream-Objekts bewegt den Stream-Zeiger auf die Position, die über die Zahl der Bytes vom Anfang des Streams spezifiziert werden muß. Diese Methode entspricht also der Standardprozedur *Seek*.

Die Funktion *GetPos* gibt einen LongInteger-Wert zurück, der die Position innerhalb des Streams speichert, auf die der Stream-Zeiger gerade zeigt.

Die Funktion *GetSize* gibt einen LongInteger-Wert zurück, der die Gesamtgröße des Stream in Byte angibt.

Die Prozedur *Truncate* löscht alle Daten nach der aktuellen Position innerhalb des Streams, wodurch der Stream-Zeiger auf das Ende des Stream zeigt.

Diese Methoden sind sicher recht nützlich, aber ein Stream, der wahlfreien Zugriff erlaubt, ist erst dann sinnvoll, wenn Sie irgendeine Art von Index außerhalb des Stream aufbauen, über den Sie dann die Position jedes Objekts im Stream erfahren können. Für diesen Zweck hervorragend geeignet ist ein *TCollection*-Objekt, und ein solches Objekt verwendet Turbo Vision zur Verwaltung von Ressourcen. Wenn Sie also einen Stream mit wahlfreiem Zugriff brauchen, sollten Sie sich die Objekte *TResourceCollection* und *TResourceFile* näher ansehen.

Daten in einem Stream speichern

In einem Stream können Sie auch Dinge abspeichern, die keine Objekte sind, Sie müssen zu diesem Zweck nur ein wenig anders vorgehen. Die Standardmethoden *Put* und *Get* setzen voraus, daß Sie Objekte speichern und lesen, die von *TObject* abstammen. Um also normale Daten in den Stream zu schreiben und daraus wieder zu laden, verwenden Sie einfach die Prozeduren *Read* und *Write*, die eine anzugebende Anzahl Bytes aus dem Stream lesen bzw. in den Stream schreiben. Auf diese Methoden stützen sich auch *Get* und *Put*, wir müssen die Anbindung an die VMT einfach umgehen, die diese Methoden herstellen.

Ein neues Stream-Objekt definieren

ist ein abstrakter Objekttyp, der erweitert werden muß, um ein verwendbares Stream-Objekt zu definieren. Sie müssen dabei beachten, daß *TStream* einerseits abstrakte Methoden enthält, die überschrieben werden müssen, andererseits auch Methoden, die sich auf diese abstrakten Methoden stützen:

Error	implementiert
Flush	abstrakt
Get	voll implementiert
GetPos	abstrakt
GetSize	implementiert, ruft aber die abstrakten Methoden *GetPos* und *SetPos* auf
Put	voll implementiert
Read	abstract
Seek	implementiert, ruft aber die abstrakte Methode *SetPos* auf
SetPos	abstrakt
Truncate	abstrakt
Write	abstrakt

Die Methoden *GetPos*, *GetSize*, *Read*, *Seek*, *Truncate*, *Write* und, wenn der Nachkomme Lese- und Schreiboperationen über einen Puffer abwickelt, auch die Methode *Flush* müssen überschrieben werden.

Wie Fehler behandelt werden

TStream definiert die Methode *Error*(*Code, Info*), die immer dann aufgerufen wird, wenn in dem Stream ein Fehler auftrat. Die Methode beschränkt sich darauf, das Feld *Status* des Streams mit einem der in der folgenden Tabelle aufgelisteten Werte zu belegen. Der Inhalt des Feldes *ErrorInfo* ist undefiniert, es sei denn, *Status* enthält den Wert *stGetError* oder den Wert *stPutError*.

stOK	Kein Fehler aufgetreten
stError	Fehler beim Dateizugriff
stInitError	Fehler bei der Stream-Initialisierung
stReadError	Versuch, über das Stream-Ende hinaus zu lesen
stWriteError	Fehler beim Schreiben in den Stream
stGetError	*Get* mit nicht registriertem Objekttyp aufgerufen
stPutError	*Put* mit nicht registriertem Objekttyp aufgerufen

Enthält *Status* den Wert *stGetError*, dann wird in *ErrorInfo* die Registrierungsnummer des nicht registrierten Objekts abgelegt. Enthält *Status* den Wert *stPutError*, dann wird in *ErrorInfo* der Offset in die VMT des Objekts eingetragen, das Sie auf dem Stream abspeichern wollten.

10. Ressourcen

Eine Ressourcen-Datei ist ein spezielles Objekt, das die Vorteile einer Kollektion mit denen eines Stream verbindet. Dadurch gibt Ihnen Turbo Vision die Möglichkeit, Objekte in einer Datei zu speichern, und Ihr Programm kann diese Objekte wieder aus dieser Datei laden. Statt sie zu initialisieren, liest Ihr Programm die benötigten Objekte aus der Ressourcen-Datei ein.

Der hinter einer Ressourcen-Datei stehende Mechanismus ist ziemlich einfach. Eine Ressourcen-Datei ist im Prinzip ein Stream mit wahlfreiem Zugriff, dessen Einträge über einen Schlüssel angesprochen werden. Dieser Schlüssel ist vom Typ String, der einen eindeutigen Namen speichert. Die Objekte, die Ihr Programm benötigt, werden sozusagen bei ihrem Namen gerufen und aus der Ressoucen-Datei geladen.

Im Gegensatz zu anderen Objekten, die Turbo Vision bereitstellt, werden Sie die für die Erstellung und Verwaltung zuständigen Objekte kaum ändern wollen. Aber Sie müssen natürlich wissen, wie Sie sie einsetzen können.

Warum Ressourcen verwenden?

Der Einsatz einer Ressourcen-Datei bringt Ihnen einige Vorteile: Sie können Ihr Programm an spezielle Anforderungen anpassen, ohne dazu den Programmcode ändern zu müssen. Sie können beispielsweise den Text in Dialog- und Menüfenstern ändern, ein *View*-Objekt in anderen Farben ausgeben lassen, ohne das Programm selbst anzutasten. Solche Änderungen betreffen nur die Ressourcen-Datei selbst.

Die Größe Ihres Programms vermindert sich, wenn Sie Ressoucen verwenden. Denn Sie können die *Init*-Konstruktoren der Objekte, die Ihr Programm benötigt, in einem separaten Programm zusammenfassen, und dieses Programm erstellt auch die Ressourcen-Datei. *Init*-Methoden sind oft sehr komplex und enthalten aufwendige Berechnungen. Statt dieser *Init*-Konstruktoren braucht Ihr Programm dann nur noch *Load*-Konstruktoren aufzunehmen, die verglichen mit ersteren relativ einfach sind. Ihr Programm wird in der Regel um 8 bis 10 Prozent kleiner, wenn es die Objekte aus einer Ressourcen-Datei einlesen kann, statt sie initialisieren zu müssen.

Ihre Programme werden auf einfache Weise »mehrsprachig«. Statt Strings im Programmcode zu übersetzen, geben Sie Ihrem Programm eine andere Ressourcen-Datei mit, und es spricht ausländisch. Die Einträge in der Ressourcen-Datei werden über ihren Namen ermittelt, was sie genau enthalten, ist dem Programm egal.

Wenn Sie verschiedene Versionen Ihres Programms erstellen wollen, die sich durch die Anzahl der verfügbaren Optionen unterscheiden, dann könnten Sie beispielsweise zwei Menübäume definieren. Der eine stellt alle Programmfunktionen zur Verfügung, der andere nur eine begrenzte Anzahl davon. Auf diese Weise brauchen Sie nicht zwei unterschiedliche Programmversionen zu schreiben und sich auch nicht darum zu sorgen, aus einer Version versehentlich die falschen Programmteile entfernt zu haben. Und Sie können die eingeschränkte Version Ihres Programms nachträglich erweitern, indem Sie es mit einer anderen Ressourcen-Datei versehen.

Kurz gesagt bieten Ihnen Ressourcen-Dateien die Möglichkeit, die Mittel der Darstellung von dem Programmcode zu trennen, der diese Darstellung ausführt. Dadurch ist es wesentlich einfacher, das Äußere Ihrer Programme zu ändern.

Was speichert eine Ressourcen-Datei?

Betrachten wir nun näher, wie Sie das Objekt *TResourceFile* einsetzen können. Wie schon erwähnt, arbeitet es zuverlässig und ist auch dem Zweck, für den es konzipiert wurde, angemessen flexibel. Sollten Sie dennoch vorhaben, es zu ändern, müssen Sie die Funktionsweise von Kollektionen und Streams gut kennen, denn das Objekt *TResourceFile* nutzt beide.

Ein *TResourceFile*-Objekt faßt eine sortierte Kollektion von Strings mit einem Stream zusammen. Die Strings in der Kollektion sind die Schlüssel, die dazu dienen, ein Objekt im Stream zu ermitteln. Die *Init*-Methode des Objekts *TResourceFile* öffnet den Stream und die Methode *Get* übernimmt einen String mit dem Namen eines Objekts und gibt dann dieses Objekt zurück.

Eine Ressourcen-Datei anlegen

Vier Schritte sind nötig, eine Ressourcen-Datei zu erstellen: Sie öffnen einen Stream, initialisieren die Datei dieses Stream, speichern darin ein oder mehrere Objekte zusammen mit ihren Schlüsseln und schließen dann die Ressourcen-Datei.

Das folgende kleine Programm legt unter dem Namen MY.REZ eine Datei an, die nur ein Objekt speichert: eine Statuszeile mit den Namen »*Waldo*« als Schlüssel.

```
program BuildResource;

uses Drivers, Objects, Views, App, Menus;

const
  cmNewDlg = 265;

type
  PHaltStream = ^THaltStream;
  THaltStream = object(TBufStream)
    procedure Error(Code, Info: Integer); virtual;
  end;

var
  MyRez  : TResourceFile;
  MyStrm : PHaltStream;

procedure  THaltStream.Error(Code, Info: Integer);
begin
  Writeln('Stream error : ', Code, ' (',Info,')');
  Halt(1);
end;

procedure CreateStatusLine;
var
  StatusLine: PStatusLine;
  R: TRect;
begin
  R.Assign(0, 24, 80, 25);
  StatusLine := New(PStatusLine, Init(R,
    NewStatusDef(0, $FFFF,
      NewStatusKey('~Alt-X~ Exit', kbAltX, cmQuit,
      NewStatusKey('~F3~ Open', kbF3, cmNewDlg,
      NewStatusKey('~F5~ Zoom', kbF5, cmZoom,
      NewStatusKey('~Alt-F3~ Close', kbAltF3, cmClose, nil)))),
      nil)));
    MyRez.Put(StatusLine, 'Waldo');
    Dispose(StatusLine, Done);
end;

begin
  RegisterMenus;
  MyStrm := New(PHaltStream, Init('MY.REZ', stCreate, 1024));
  MyRez.Init(MyStrm);
  CreateStatusLine;
  MyRez.Done;
end.
```

Objekte aus einer Ressourcen-Datei lesen

Die Statuszeile, die wir gerade unter dem Namen »*Waldo*« in einer Ressourcen-Datei haben speichern lassen, wollen wir nun wieder einlesen. Dazu rufen wir die Methode *Get* auf und übergeben ihr in Form eines Strings diesen Namen. *Get* liefert einen Zeiger des Typs *PObject* zurück, der das Objekt Statuszeile referenziert:

```
program MyApp;

uses Objects, Drivers, Views, Menus, Dialogs, App;

var
  MyRez: TResourceFile;

type
  PMyApp = ^TMyApp;
  TMyApp = object(TApplication)
    constructor Init;
    procedure InitStatusLine; virtual;
  end;

constructor TMyApp.Init;
var
  S: PStream;
  Event: TEvent;
const
  MyRezFileName: FNameStr = 'MY.REZ';
begin
  MyRez.Init(New(PBufStream, Init(MyRezFileName, stOpen, 1024)));
  if MyRez.Stream^.Status <> 0 then Halt(1);
  TApplication.Init;
end;

procedure TMyApp.InitStatusLine;
begin
  StatusLine := PStatusLine(MyRez.Get('Waldo'));
end;
```

Wenn Sie ein Objekt aus einer Ressourcen-Datei anfordern, müssen Sie mit der Möglichkeit rechnen, einen **nil**-Zeiger zurückzuerhalten. Wenn das Objekt nicht gefunden wurde, weil der übergebene Schlüssel ungültig war, also kein Objekt auf diesen Namen hört, dann gibt *Get* **nil** zurück. Haben Sie aber Ihr Programm ausgetestet und eventuelle Fehler beseitigt, wird auch dies kein Problem mehr sein.

Sie können ein Objekt auch mehrmals aus der Ressourcen-Datei einlesen. Unwahrscheinlich, daß dies für die Statuszeile unseres Beispiels erforderlich sein sollte. Aber für ein Dialogfenster beispielsweise kann sich durchaus die Notwendigkeit ergeben, daß es während eines Programmlaufs mehrere Male

einzulesen ist. Das Objekt *TResourceFile* liefert einen Eintrag in der Datei immer dann zurück, wenn es dazu aufgefordert wird – auch mehrmals.

Bei einer relativ langsamen Ein- und Ausgabe auf Diskette kann dies natürlich zu Problemen führen, auch wenn *TResourceFile* diese Operationen über einen Puffer durchführt. Sie können diesen Puffer angemessen vergrößern oder, wenn EMS-Speicher zur Verfügung steht, den Ressourcen-Stream in einen EMS-Stream kopieren.

Strings in Listen

Turbo Vision definiert zwei Objekte, deren Spezialität darin besteht, Listen von Strings aufbauen und verwalten zu können. Auch solche Listen können in einer Ressourcen-Datei gespeichert werden. Es sind jedoch Nummern, nicht die Strings selbst, über die die Elemente einer String-Liste identifiziert werden. Ansonsten bieten auch diese Objekte die Vorteile, auf die wir bei der Besprechung der Ressourcen, die Objekte abspeichern, hingewiesen haben.

Beipielsweise könnten Sie Fehlermeldungen, die Ihr Programm gegebenenfalls ausgeben muß, in einer solchen Liste speichern. Ihr Programm kann dann über die Nummer des Fehlers den passenden String aus der Liste anfordern. Programmversionen für verschiedene Länder unterscheiden sich dann nur darin, daß ihre String-Liste anderssprachige Meldungen speichert.

Das Objekt *TStringList* ist naturgemäß nicht sonderlich flexibel, bietet aber die Möglichkeit schnellen Zugriffs und ist für bestimmte Zwecke völlig ausreichend.

Das Objekt *TStringList* greift auf eine String-Liste zu, die von dem Objekt *TStrListMaker* erstellt wurde. Wie für alle Objekte definiert Turbo Vision auch für diese einen Registrierungsrecord, mittels dessen sie zur Verwendung mit einem Stream angemeldet werden. Allerdings besitzen beide Objekte die gleiche Registrierungsnummer, können also nicht gleichzeitig in einem Programm verwendet werden. Diese Objekte werden deshalb auch nicht automatisch von Turbo Vision angemeldet. Sie müssen vielmehr das Objekt, das Sie verwenden, über *RegisterType* selbst anmelden.

Das Objekt TStringList hat auch keinen *Init*- sondern nur einen *Load*-Konstruktor, denn die String-Liste ist schließlich in einer Ressourcen-Datei gespeichert, auf die *TStringList* nur lesend zugreift. Daher verfügt dieses Objekt auch nur über eine *Get*-Methode und definiert keine *Put*-Prozedur.

Eine Liste von Strings erstellen

Das Objekt TStrListMaker erstellt eine String-Liste und speichert sie in einer Ressourcen-Datei, auf die *TStringList* zugreifen kann. Auf die Datei greift das Objekt *TStrListMaker* nur schreibend zu, und besitzt deshalb auch nur eine *Put*-Methode aber keine Funktion *Get*. Sie können dieses Objekt also nur dazu

benutzen, eine String-Liste zu initialisieren, einen Eintrag nach dem andern in ihr abzulegen und schließlich diese Liste in einem Stream zu speichern.

11. Tips und Tricks

Diese Kapitel enthält einige zusätzliche Vorschläge, wie Sie Turbo Vision noch effizienter nutzen können. Da objektorientierte Programmierung und die Erstellung ereignisgesteuerter Programme auch für erfahrene Programmierer relativ neue Konzepte darstellen, wollen wir einige Hinweise zur Nutzung dieser neuen Paradigmen geben.

Die Fehlersuche in Turbo-Vision-Programmen

Sie haben vielleicht versucht, mittels des Debuggers in der integrierten Entwicklungsumgebung eines der mitgelieferten Demo-Programme schrittweise ausführen zu lassen und Sie mußten feststellen, daß Sie mit diesem Verfahren nicht sehr weit kamen. Turbo-Vision-Programme sind ereignisgesteuert. Die meiste Zeit durchläuft ein solches Programm eine Schleife in *TGroup.Execute*, darauf wartend, daß ein Ereignis eintritt. Die Verfolgung des Programmablaufs mittels dessen schrittweiser Ausführung ist daher nicht sehr erhellend.

Wenn Sie einen Fehler in einem Turbo-Vision-Programm aufspüren müssen, dann gibt es nur ein Rezept: Breakpoints setzen.

Betrachten wir, wie ein Breakpoint am richtigen Ort das Problem lösen kann.

Wo ist es denn abgeblieben?

Ein Problem, das in Ihrem Programm auftreten könnte, ist, daß ein bestimmter Programmteil nie ausgeführt wird. Sie klicken einen Eintrag in der Statuszeile an oder Sie selektieren eine Menüoption, und nichts passiert. Sie wissen aber, daß sich ein Fenster hätte öffnen müssen.

Möglicherweise versuchen Sie jetzt, Ihr Programm schrittweise auszuführen, um, wenn der betreffende Befehl auftritt, herausfinden zu können, welcher Programmteil statt des erwarteten ausgeführt wird. Dieses Vorgehen

bringt nicht viel. Sie lassen Ihr Programm schrittweise ausführen, und früher oder später finden Sie sich genau dort wieder, wo Sie angefangen haben.

In dieser Situation gehen Sie am besten so vor, daß Sie einen Breakpoint am Beginn der HandleEvent-Methode setzen, die den betreffenden Programmteil aufrufen sollte. Wenn die Programmausführung dann abgebrochen wird, untersuchen Sie zunächst den gerade bearbeiteten Ereignis-Record und vergewissern sich, daß er tatsächlich das erwartete Ereignis speichert. Ab diesem Punkt können Sie das Programm dann schrittweise ausführen lassen, denn die Programmteile, die für von Ihnen definierte Ereignisse zuständig sind, haben Sie geschrieben. Daher können Sie die Ausführung dieser Anweisungen auch verfolgen.

Hinter der Maske versteckt

Bedenken Sie auch, daß Ihr Objekt, das ein bestimmtes Ereignis bearbeiten soll, aus verschiedenen Gründen dieses Ereignis nie zu sehen bekommt. Der erste und einfachste Grund dafür: Sie haben vergessen, Ihrem Objekt mitzuteilen, daß es für diese Ereignisart zuständig ist. Sie müssen in dem Feld EventMask das Bit setzen, das der Art des Ereignisses entspricht. Denn andernfalls wird das Objekt diesen Ereignissen die kalte Schulter zeigen.

Geklaute Ereignisse

Eine zweite Möglichkeit, weshalb ein Ereignis nie zu dem Objekt gelangt, das dieses Ereignis abarbeiten soll, besteht darin, daß es ein anderes Objekt weggeschnappt hat. Das Ereignis wird von diesem Objekt bearbeitet und als erledigt markiert.

Auch dafür kann es verschiedene Gründe geben. Vielleicht haben Sie zwei verschiedenen Befehlskonstanten den gleichen Wert zugeordnet. Dann wird eine ganz andere als die erwartete Aktion ausgeführt. Daher sollten Sie über die Definition von Befehlen immer genau Buch führen. Das ist ganz besonders wichtig dann, wenn Sie in Ihrem Programm bestehende Module einbinden.

Mißtrauen Sie den Vorfahren

Überprüfen Sie auch, ob nicht die *HandleEvent*-Methode eines Vorfahren Ihres Objekts das Ereignis bearbeitet hat. Abgeleitete Objekttypen rufen oft die geerbte *HandleEvent*-Methode auf, um einen Großteil der eingehenden Ereignisse abarbeiten zu lassen. Überprüfen Sie daher, ob das betreffende Ereignis nach dem Aufruf der geerbten Methode noch vorliegt.

Das haben wir nicht erwartet

Vielleicht öffnet sich das Fenster auch, aber es ist nur Müll in ihm zu sehen oder etwas, das zu sehen Sie nicht erwartet haben. Das Ereignis wurde korrekt entgegengenommen, aber die Methode, die es bearbeitet, ist entweder nicht korrekt oder wurde überschrieben. In diesem Fall setzen Sie natürlich einen Breakpoint in der Routine, die aufgerufen wird, wenn das betreffende Ereignis

eintritt. Wird das Programm dann unterbrochen, können Sie es schrittweise ausführen lassen und verfolgen, was geschieht.

Das Programm steht

und nichts geht mehr. Fehler mit solchen Auswirkungen sind sicher schwer zu finden, aber auch sie können aufgespürt werden. Sie können zunächst, wie oben diskutiert, einige Breakpoints setzen, um den Programmteil möglichst genau einzugrenzen, dessen Ausführung Ihr Programm zum Stehen bringt. Als zweites sollten Sie nach Zeigern suchen, die zweimal freigegeben werden. Das kann passieren, wenn der von einer Gruppe belegte Speicher freigegeben wird, was automatisch auch den Speicher aller Subviews dieser Gruppe einschließt. Wenn Sie dann die Destruktor-Methode einer dieser *View*-Objekte direkt aufrufen, wird Ihr Programm hängenbleiben. Der folgende Programmauschnitt zeigt einen solchen Fehler (weshalb Sie diese Anweisungen auch nicht ausführen lassen dürfen):

```
var
  Bruce, Pizza: PGroup;
  R: TRect;
begin
  R.Assign(5, 5, 10, 10);
  Pizza := New(PGroup, Init(R));
  R.Assign(10, 10, 10, 20);
  Bruce := New(PGroup, Init(R));
  Bruce^.Insert(Pizza); { Pizza in Gruppe Bruce einf¡gen }
  Dispose(Bruce, Done);   { Bruce und Subviews freigeben }
  Dispose(Pizza, Done);     { ab hier steht ihr Computer }
end;
```

Der Aufruf des Destruktors von *Bruce* führt zur Freigabe des von der Gruppe **und** ihrer Subviews belegten Speichers. Daher führt der Aufruf der Destruktor-Methode von *Pizza* zum Absturz des Programms.

Auch wenn aus einem Stream Daten in den falschen Objekttyp eingelesen werden oder der Typ der in einer Kollektion gespeicherten Daten falsch interpretiert wird, kann ein Absturz des Programms die Folge sein.

Turbo Vision in bestehende Programme einbinden

Auf den ersten Blick mag Turbo Vision nur als eine neue Benutzeroberfläche erscheinen. Was liegt also näher, als diese Benutzeroberfläche über schon bestehende Programme zu legen? Sie werden an dieser Art Kulissenschieberei wenig Freude haben. Turbo Vision ist mehr als nur eine Programmoberfläche, sondern übernimmt die bürokratischen Aufgaben in Ihren Pro-

grammen. Turbo-Vision-Programme sind objektorientiert und ereignisgesteuert. Nur wenige bestehende Programme werden sich ohne massive Änderungen diesen neuen Paradigmen der Programmierung fügen.

Das Programm durchforsten

Es gibt einen einfacheren Weg. Sie wissen, daß die Besonderheiten verschiedener Turbo-Vision-Programme an den Methoden *Init*, *Draw* und *HandleEvent* von *TApplication* anknüpfen. Wenn Sie also ein bestehendes Programm Turbo-Vision-tauglich machen wollen, dann schreiben Sie am besten mit Hilfe von Turbo Vision eine Programmoberfläche, die dem, was Ihr Programm bereitstellt, entspricht. Dann binden Sie Ihre Routinen Stück für Stück in diese neue Oberfläche ein. Sie werden feststellen, daß Ihr Programmcode in den Methoden *Init*, *Draw* und *HandleEvent* neuer *View*-Objekte seinen Platz finden wird.

Zunächst aber müssen Sie in Ihren Programmen die Bürokratie von der Produktion trennen. Sie müssen also jene Teile, die den produktiven Teil Ihres Programms ausmachen, von den Teilen trennen, die für die Ausgabe und Verwaltung Ihrer Oberfläche zuständig sind. Sie werden sicher mit Erstaunen feststellen, welch großer Teil Ihres Programms nur diesen bürokratischen Aufgaben gewidmet war. Aber das übernimmt jetzt Turbo Vision.

Natürlich werden Sie auch einige Teile neu schreiben. Denn Sie müssen ja zum Beispiel Ihre Objekte in die Lage versetzen, sich selber auf dem Bildschirm darstellen zu können. Die meiste Arbeit aber werden Sie damit haben, nicht mehr gebrauchten Programmcode zu entfernen.

Wichtiger aber ist, die Organisation Ihres Programms zu überdenken und sie gegebenenfalls neu zu strukturieren. Vielleicht haben Sie Ihr Programm so organisiert, daß jeder Programmteil in der Lage war, bei Bedarf mit jedem anderen Daten auszutauschen. Das ist in einem objektorientierten Programm nicht mehr so leicht möglich. Betrachten wir dazu ein Ihnen bekanntes Beispiel.

Ein Programm organisieren

Sie kennen die integrierte Entwicklungsumgebung von Turbo Pascal und Sie wissen auch, daß diese Entwicklungsumgebung auf Turbo Vision aufbaut. Wie ist nun dieses Programm intern organisiert, wo liegt der Dreh- und Angelpunkt? In der Entwicklungsumgebung können Sie Programmtexte editieren, Sie können ein Programm compilieren und Sie können eventuelle Fehler mittels des integrierten Debuggers lokalisieren. Die meiste Zeit werden Sie jedoch mit dem Editor arbeiten. Was liegt nun näher, als den Rest des Programms um diesen Editor herum zu organisieren?

Aber das wäre ein falscher Ansatzpunkt. Denn was muß eigentlich der

Editor von der Existenz des Compilers wissen? Was umgekehrt der Compiler von der Existenz des Editors? Viel, werden Sie sagen. Denn schließlich muß der eine Programmteil ja mit dem anderen kommunizieren. Trifft der Compiler beispielsweise auf einen Fehler in einer Programmdatei, muß er dem Editor diesen Tatbestand mitteilen.

Nein, muß er nicht. Der Compiler liefert lediglich eine Meldung über das Resultat seiner Bemühungen zurück. Stößt er auf einen Fehler, dann informiert diese Meldung auch über die Art des Fehlers, den Namen der Datei und die Nummer der Zeile. Diese Meldung wird von der Entwicklungsumgebung entgegengenommen, nicht vom Editor. Und die Entwicklungsumgebung überprüft, ob die betroffenene Datei bereits im Editor geladen und ein entsprechendes Fenster geöffnet ist. Sollte dies nicht der Fall sein, öffnet sie ein Fenster, übergibt dem Editor den Namen der Datei, die Nummer der Zeile und eine Fehlermeldung.

Sie sehen, es ist die Entwicklungsumgebung, die die Zusammenarbeit der verschiedenen Programmteile organisiert. Sie nimmt Meldungen entgegen und veranlaßt gegebenenfalls, daß andere Programmteile mit ihrer Arbeit beginnen. Sie und nicht der Editor ist das Zentrum des Programms, und nur über sie und von ihr kontrolliert kommunizieren die restlichen Programmteile miteinander.

Jeder Programmteil ist etwas übertrieben ausgedrückt autark. Er führt eine bestimmte Aktion durch und berichtet über das Ergebnis der Aktion. Die Entwicklungsumgebung fordert beispielsweise den Compiler auf, eine Datei zu compilieren, und nimmt den Bericht über den Abschluß dieser Aktion entgegen. Welche Maßnahmen dann zu ergreifen sind, muß die Entwicklungsumgebung entscheiden, der Compiler interessiert sich dafür nicht. Die Organisation eines Programms über einen solchen »Verwalter« führt dazu, das Programm robuster zu machen, und hat den willkommenen Nebeneffekt, daß die einzelnen Programmteile leichter auch in anderen Programmen einsetzbar sind.

Bits manipulieren

Die *View*-Objekte definieren verschiedene Felder, deren einzelne Bits für Turbo Vision eine bestimmte Bedeutung haben. Ein Bit kann den Wert 1 oder 0 haben, ist also *gesetzt* oder *gelöscht*. Es wird dazu benutzt, anzuzeigen, ob eine bestimmte Eigenschaft vorhanden ist oder nicht. Für ein auf diese Weise interpretiertes Bit hat sich der englische Begriff *Flag* eingebürgert.

Jedes von *TView* abgeleitete Objekt verfügt zum Beispiel über das Feld *Options*. Dieses Feld ist vom Typ Word, und jedes einzelne Bit dieses Word hat für Turbo Vision eine bestimmte Bedeutung. Der Eintrag »ofXXXX-Konstanten« in Kapitel 15, »Außerhalb der Objekte« beschreibt die Bedeutung der einzelnen Bits im Feld *Options*. Dort finden Sie auch eine Abbildung, die jedes Bit als kleines Kästchen darstellt. Das von links gesehen ersten Kästchen ist mit »msb«, das von rechts gesehen erste Kästchen mit »lsb« markiert. Die

Abkürzung »msb« steht für *most significant bit*, bezeichnet also das höherwertigste Bit in *Options*; »lsb« steht für *least significant bit* und bezeichnet das niedrigstwertige Bit dieses Feldes.

Flags

Das vierte Bit zum Beispiel hat in Turbo Vision den »Namen« *ofFramed*. Hat es den Wert 1 – ist also *gesetzt* – dann verfügt das *View*-Objekt über einen sichtbaren Rahmen. Hat es den Wert 0 – ist also *gelöscht* – besitzt das Objekt keinen Rahmen. Im allgemeinen müssen Sie sich nicht darum kümmern, welche Bits in *Options* gesetzt und welche gelöscht sind. Wenn Sie aber vorhaben, selbst solche Flags zu definieren, müssen Sie einerseits darauf achten, daß diese Flags eindeutig sind, und andererseits wissen, welche Bits Ihnen noch zur Verfügung stehen. Die sechs höherwertigen Bits in Options beipielsweise werden von Turbo Vision nicht benutzt. Ihre *View*-Objekte können jedes dieser Bits auf eine von Ihnen festgelegte Weise interpretieren.

Bitmasken

Über eine *Bitmaske* können Sie auf einfache Weise eine ganze Gruppe von Bits ansprechen. Turbo Vision zum Beispiel definiert Bitmasken zur Unterscheidung der verschiedenen Ereigniskategorien. Die Bitmaske *evMouse* faßt alle vier Bits zusammen, von denen jedes ein spezielles Mausereignis bezeichnet. Wenn also ein *View*-Objekt überprüfen muß, ob ein Mausereignis vorliegt, kann es über diese Bitmaske alle vier relevanten Bits erfassen, statt sie einzeln überprüfen zu müssen.

Bitoperationen

Zur Manipulation einzelner Bits stellt Turbo Pascal einige nützliche Operatoren bereit. Wir zeigen Ihnen, welche davon Sie für welchen Zweck einsetzen können.

Ein Bit setzen

Um ein Bit zu setzen, benutzen Sie den **or**-Operator. Über die folgende Anweisung wird das Bit *ofPostProcess* im Feld *Options* des *View*-Objekts *MyButton* gesetzt:

```
MyButton.Options := MyButton.Options or ofPostProcess
```

Benutzen Sie dazu auf keinen Fall den Additionsoperator, es sei denn, Sie wissen genau was Sie tun. Die Anweisung

```
MyButton.Options := MyButton.Options + ofPostProcess
```

funktioniert nur dann, wenn das Bit *ofPostProcess* gelöscht war. War es aber vor der Ausführung der Addition gesetzt, wird das Bit gelöscht und das

folgende Bit (*ofBuffered*) gesetzt, wenn es gelöscht oder umgekehrt gelöscht, wenn es gesetzt war.

Verwenden Sie also den **or**-Operator, um ein Bit zu setzen. Über diesen Operator können Sie auch mehr als nur ein Bit setzen. Die folgende Anweisung setzt gleichzeitig zwei Bits im Feld *GrowMode* des Objekts *MyScroller*:

```
MyScroller.GrowMode := MyScroller.GrowMode
  or (gfGrowHiX + gfGrowHiY);
```

Ein Bit löschen

Um ein Bit zu löschen, benutzen Sie eine Kombination der beiden Bit-Operatoren **and** und **not**. Um das Bit *dmLimitLoX* im Feld *DragMode* des Objekts *ALabel* zu löschen, benutzen Sie die Anweisung

```
ALabel.DragMode := ALabel.DragMode and not dmLimitLoX;
```

Auch mit diesen Operatoren ist es möglich, über eine Anweisung mehrere Bits zu löschen.

Ein Bit überprüfen

Oft müssen Sie überprüfen, ob in einem Feld ein bestimmtes Bit gesetzt ist oder nicht. Für diesen Zweck ist **and** der passende Operator. Um festzustellen, ob das Fenster *AWindow* geordnet neben anderen Objekten dargestellt werden darf, müssen Sie überprüfen, ob im Feld *Options* dieses Objekts das Bit *ofTileable* gesetzt ist:

```
if AWindow.Options and ofTileable = ofTileable then ...
```

Bitmasken einsetzen

Mittels einer Bitmaske können Sie mit dem Operator **and** auch feststellen, ob in einem Feld eines oder mehrere der in der Bitmaske gesetzten Bits ebenfalls gesetzt ist. Wollen Sie beispielsweise feststellen, ob ein Eregnisrecord ein Mausereignis speichert, dann verwenden Sie die Anweisung

```
if Event.What and evMouse <> 0 then ...
```

Zusammenfassung

Die folgende Liste faßt diese Bitoperationen zusammen:

- Ein Bit setzen:

  ```
  Feld := Feld or Flag;
  ```

- Ein Bit löschen

  *EventMask*Feld := Feld **and not** Flag;

- Überprüfen, ob ein Bit gesetzt ist:

  ```
  if Feld and Flag = Flag then ...
  ```

- Überprüfen, ob ein in einer Bitmaske gesetztes Bit auch in einem Feld gesetzt ist:

```
if Feld and Bitmaske <> 0 then ...
```

12. Benutzung der Referenzkapitel

Die Einführung in Turbo Vision und die Besprechung seiner wichtigsten Komponenten ist damit abgeschlossen. Die nun noch folgenden drei Kapitel sind für Referenzzwecke gedacht. Den gewichtigsten Teil macht die Auflistung der in Turbo Vison definierten Standardobjekte aus. Daneben finden Sie aber auch die außerhalb der Objekte definierten Prozeduren und Funktionen, Datentypen, Variablen und Konstanten. Außerdem geben wir auch einen Überblick über das, was die Units enthalten, die zu Turbo Vision gehören.

Was Sie wo finden

Kapitel 13 »Was die Units exportieren« ist eine nach den Units geordnete Auflistung aller Objekte, Routinen, Datentypen, Variablen und Konstanten, die von den zu Turbo Vision gehörenden Units exportiert werden. Dadurch können Sie ermitteln, welche Unit Sie einbinden müssen, um ein bestimmtes Objekt verwenden zu können.

Kapitel 14 »Verzeichnis der Objekte« führt in alphabetischer Folge alle in Turbo Vision definierten Objekte, einschließlich ihrer Felder und Methoden auf. Jedes Feld und jede Methode ist mit einer ausführlichen Erläuterung versehen.

Kapitel 15 »Außerhalb der Objekte« führt in alphabetischer Folge und mit Erläuterungen versehen alle von den zu Turbo Vision gehörenden Prozeduren, Funktionen, Datentypen, Variablen und Konstanten auf. Kurz gesagt finden Sie hier alles, was kein Objekt ist.

Beachten Sie, daß wir in der Auflistung der Objekte nur die jeweils neu definierten Felder und Methoden beschreiben. Aber ein abgeleitetes Objekt verfügt auch über die Felder und Methoden seiner Vorfahren. Wenn Sie also Informationen zu einer bestimmten Methode eines Objekts suchen, kann es sein, daß Sie diese Information nicht in dem Eintrag finden, der diesem Objekt gewidmet ist. Überprüfen Sie dann den Eintrag des Vorfahren darauf, ob er die gesuchte Methode beschreibt. Um Ihnen die Situierung eines Objekts innerhalb der von Turbo Vision definierten Hierarchie zu erleichtern, finden Sie am Beginn jedes Eintrags eine Grafik, die den Vorfahren des Objekts und gegebenenfalls dessen Nachfahren zeigt.

Namens-konventionen

Sie werden bemerkt haben, daß Turbo Vision bei der Vergabe von Bezeichnern bestimmten Konventionen folgt. Ein Objekt, der Zeiger auf dieses Objekt, der zugeordnete Registrierungsrecord und gegebenenfalls die Farbpalette tragen alle fast den gleichen Namen. Es unterscheiden sich lediglich die ersten Buchstaben:

- Der Name eines Objekttyps beginnt mit einem **T**: *TObject*.
- Der Name eines Zeigers auf dieses Objekt beginnt mit einem **P:** *PObject* = ^*TObject*.
- Der Name des Records zur Registrierung dieses Objekts beginnt mit einem **R**: *RObject*.
- Der Name der Farbpalette beginnt mit einem **C**: *CObject*.

Die Namen der in Turbo Vision definierten Konstanten beginnen mit einem zwei Zeichen langen Präfix, das die Bedeutung der jeweiligen Konstante andeutet:

Präfix	Bedeutung	Beispiel
ap	Farbpalette des Programms	*apColor*
bf	Schalter-Flags	*bfNormal*
cm	Befehl	*cmQuit*
co	Fehlercode Kollektion	*coOverflow*
dm	Bewegungsmodus von *View*-Objekten	*dmDragGrow*
ev	Ereignisarten	*evMouseDown*
gf	Größenveränderungsmodus	*gfGrowLoX*
hc	Hilfenummern	*hcNoContext*
kb	Tasten	*kbAltX*
mb	Mausknöpfe	*mbLeftButton*
of	Flag für *Options*	*ofTopSelect*
sb	Rollbalken-Bestandteile	*sbLeftArrow*
sf	Flags für *State*	*sfVisible*
sm	Bildschirmmodus	*smMono*
st	Zugriffs-/Fehlercode Stream	*stOK*
wf	Flags für Fenster	*wfMove*
wn	Fensternummer	*wnNoNumber*
wp	Farbpalette für Fenster	*wpBlueWindow*

13. Was die Units exportieren

Dieses Kapitel beschreibt kurz, was die Units enthalten, die zu Turbo Vision gehören. Zu Beginn geben wir einen Überblick über diese Units, dann gehen wir auf deren Inhalt näher ein.

Unit	Inhalt
App	Alle zum Schreiben ereignisgesteuerter Programme notwendigen Objekttypen
Dialogs	Elemente für den Aufbau von Dialogfenstern
Drivers	Maus- und Tastatur-Unterstützung, Fehlerbehandlung usw.
HistList	Eingabeaufzeichnungslisten
Memory	Speicherverwaltung
Menus	Objekte zur Ausgabe von Menüs und der Statuszeile
Objects	Definition von Basisobjekten, Streams, Kollektionen und Ressourcen
TextView	Spezialisierte *View*-Objekte zur Ausgabe von Text
Views	Objekte zur Nutzung von Fenstern in Ihren Programmen: Fenster, Rahmen, Rollbalken usw.

Unit Objects

Diese Unit enthält die Definitionen der Basisobjekte von Turbo Vision, eingeschlossen das Objekt *TObject*, das Fundament der Objekthierarchie, und die stummen Objekte: Streams, Kollektionen und Ressourcen.

Typen

Typkonvertierung

Typ	Funktion
FNameStr	Stringtyp für einen DOS-Dateinamen
LongRec	Zugriff auf nieder- und höherwertiges Word eines LongInt
PChar	Zeiger auf ein Zeichen
PString	Zeiger auf einen String
PtrRec	Zugriff auf Segment und Offset eines Zeigers
TByteArray	Byte-Array
TWordArray	Word-Array
WordRec	Zugriff auf nieder- und höherwertiges Byte eines Word

Übrige Typen

Typ	Funktion
TBufStream	Gepufferter DOS-Stream
TCollection	Polymorphes Array
TDosStream	Stream, der DOS-Datei verwendet
TEmsStream	Stream, der EMS-Speicher verwendet
TItemsList	Zeiger-Array, von Kollektionen genutzt
TObject	Basis-Objekt der Turbo-Vision-Hierarchie
TPoint	Objekt, das einen Punkt bezeichnet
TRect	Objekt, das einen rechteckigen Bereich definiert
TResourceCollection	Spezialisierte Kollektion für Ressourcen
TResourceFile	Objekt, das Ressourcen in einer Datei speichert
TSortedCollection	Kollektion, die ihre Einträge sortiert
TStream	Basisobjekt zur Ableitung von Streams
TStreamRec	Registrierungsrecord zur Anmeldung eines *Objekttyps*
TStrIndex	Array von *TStrIndexRec*
TStrIndexRec	Record mit Indices auf Strings; von *TStrIndex* benutzt
TStringCollection	Sortierte Kollektion für Strings
TStringList	Objekt, das Strings aus einer Ressourcen-Datei liest
TStrListMaker	Spezielles Objekt, das Strings in einer Ressourcen-Datei speichert

Konstanten

Zugriffsmodus für Streams

Konstante	Wert	Bedeutung
stCreate	$3C00;	Neue Datei anlegen
stOpenRead	$3D00;	Nur Lesen möglich
stOpenWrite	$3D01;	Nur Schreiben möglich
stOpen	$3D02;	Lesen und Schreiben möglich

Stream-Fehlercodes

Fehlercode	Wert	Bedeutung
stOk	0	Kein Fehler aufgetreten
stError	-1	Zugriffsfehler
stInitError	-2	Initialisierung des Stream fehlgeschlagen
stReadError	-3	Versuch, über das Ende des Stream hinaus zu lesen
stWriteError	-4	Fehler beim Schreiben in den Stream
stGetError	-5	*Get* mit nicht registriertem Stream
stPutError	-5	*Put* mit nicht registriertem Stream

Maximale Größe einer Kollektion

Konstante	Wert	Bedeutung
MaxCollectionSize	16380	Maximalzahl der Einträge in einer Kollektion

Fehler in einer Kollektion

Fehlercode	Wert	Bedeutung
coIndexError	-1	Index außerhalb des zulässigen Bereichs
coOverflow	-2	Aufnahme des Eintrags nicht mehr möglich

Variablen

Variable	Typ	Anfangswert	Bedeutung
EmsCurHandle	Word	$FFFF	Aktueller EMS-Handle
EmsCurPage	Word	$FFFF	Aktuelle EMS-Page

Prozeduren und Funktionen

Prozedur	Operation
Abstract	Löst bei Aufruf einer abstrakten Methode einen Laufzeitfehler aus
DisposeStr	Freigabe des über *NewStr* reservierten Speichers für einen String
RegisterObjects	Registriert alle Objekte dieser Unit für die Arbeit mit einem Stream
RegisterType	Registriert einen Objekttyp für Verwendung mit einem Stream

Funktion	Operation
LongDiv	Dividiert einen LongInt durch einen Integer
LongMul	Gibt einen LongInt als Resultat der Multiplikation zweier Integer zurück
NewStr	Reserviert Speicher für einen String

Unit Views

Diese Unit bietet die Grundlage für alle Objekte, die sich auf dem Bildschirm darstellen. Dies gilt für die abstrakten Objekttypen *TView* und *TGroup*. Aber auch komplexere *View*-Objekte sind in dieser Unit zu finden, zum Beipiel Rollbalken und Rahmen für Bildschirmfenster. Weitere komplexe *View*-Objekte stellen die Units *Dialogs* und *TextView* bereit.

Typen

Typ	Benutzung
TCommandSet	Datentyp zur Speicherung von Befehlen
TDrawBuffer	Puffer, der von *Draw*-Methoden benutzt wird
TFrame	Objekt, das Fenster mit einem Rahmen versieht
TGroup	Abstraktes Objekt für komplexe Darstellungen
TListViewer	Basis-Objekt zur Ausgabe von Listen
TPalette	Datentyp für Definition von Farbpaletten
TScrollBar	Objekt zur Ausgabe eines Rollbalkens
TScrollChars	Zeichen, verwendet bei der Darstellung eines Rollbalkens
TScroller	Basis-Objekt für Textfenster
TTitleStr	Datentyp für Fenstertitel
TVideoBuf	Bildschirmpuffer
TView	Objekt, von dem alle sichtbaren Objekte abgeleitet sind
TWindow	Basis-Objekt für in ihrer Größe änderbare Fenster

Konstanten

Allgemeine Konstanten

Konstante	Wert	Bedeutung
hcNoContext	0	Neutrale Hilfenummer
hcDragging	1	Hilfenummer, während das Objekt über den Bildschirm bewegt wird
MaxViewWidth	132	Maximale Breite eines *View*-Objekts
wnNoNumber	0	Fenster hat keine Nummer

Bitmasken für TView.State

Konstante	Wert	Bedeutung
sfVisible	$0001	Objekt ist sichtbar
sfCursorVis	$0002	Cursor ist sichtbar
sfCursorIns	$0004	Blockcursor als Zeichen für aktiven Einfügemodus
sfShadow	$0008	Objekt hat einen Schatten
sfActive	$0010	Objekt ist aktiv oder gehört zur aktiven Gruppe
sfSelected	$0020	Objekt ist das selektierte *View*-Objekt der Gruppe
sfFocused	$0040	Objekt ist fokussiert
sfDragging	$0080	Objekt wird über den Bildschirm bewegt
sfDisabled	$0100	Objekt ist deaktiviert
sfModal	$0200	Objekt ist modales *View*-Objekt
sfExposed	$0800	Objekt gehört zu *TApplication*

Bitmasken für TView.Option

Konstante	Wert	Bedeutung
ofSelectable	$0001	Objekt kann selektiert werden
ofTopSelect	$0002	Wenn selektiert, wird das Objekt vor allen andern Objekten auf dem Bildschirm dargestellt
ofFirstClick	$0004	Mausklick selektiert *und* führt Aktion aus
ofFramed	$0008	Objekt besitzt einen sichtbaren Rahmen
ofPreProcess	$0010	Objekt erhält fokussierte Events vor dem fokussierten *View*-Objekt
ofPostProcess	$0020	Objekt erhält fokussierte Events nach dem fokussierten *View*-Objekt
ofBuffered	$0040	Der Gruppe wird ein Puffer (für Bildschirmausgabe) zugeordnet
ofTileable	$0080	Objekt kann ausgerichtet neben anderen Objekten angeordnet werden
ofCenterX	$0100	Objekt in X-Richtung innerhalb der Gruppe zentrieren
ofCenterY	$0200	Objekt in Y-Richtung innerhalb der Gruppe zentrieren
ofCentered	$0300	Objekt in X- und Y-Richtung innerhalb der Gruppe zentrieren

Bitmasken für TView.GrowMode

Konstante	Wert	Bedeutung
gfGrowLoX	$01	Linke Seite folgt der rechten Seite der Gruppe
gfGrowLoY	$02	Obere Seite folgt der unteren Seite der Gruppe
gfGrowHiX	$04	Rechte Seite folgt der linken Seite der Gruppe
gfGrowLoY	$08	Untere Seite folgt der unteren Seite der Gruppe
gfGrowAll	$0F	Objekt folgt der unteren rechten Ecke der Gruppe
gfGrowRel	$10	Verändert sich relativ zur Größenveränderung der Gruppe

Bitmasken für TView.DragMode

Konstante	Wert	Bedeutung
dmDragMove	$01	Objekt kann bewegt werden
dmDragGrow	$02	Objekt kann seine Größe ändern
dmLimitLoX	$10	Linke Seite darf nicht über *Limits* hinausbewegt werden
dmLimitLoY	$20	Obere Seite darf nicht über *Limits* hinausbewegt werden
dmLimitHiX	$40	Rechte Seite darf nicht über *Limits* hinausbewegt werden
dmLimitHiY	$80	Untere Seite darf nicht über *Limits* hinausbewegt werden
dmLimitAll	$F0	Kein Teil des Objekts darf über *Limits* hinausbewegt werden

Teile eines Rollbalkens

Konstante	Wert	Bedeutung
sbLeftArrow	0	Linkes Pfeilsymbol eines horizontalen Rollbalkens
sbRightArrow	1	Rechtes Pfeilsymbol eines horizontalen Rollbalkens
sbPageLeft	2	Linker Rollbereich eines horizontalen Rollbalkens
sbPageRight	3	Rechter Rollbereich eines horizontalen Rollbalkens
sbUpArrow	4	Oberes Pfeilsymbol eines vertikalen Rollbalkens
sbDownArrow	5	Unteres Pfeilsymbol eines vertikalen Rollbalkens

Konstante	*Wert*	*Bedeutung*
sbPageUp	6	Oberer Rollbereich eines vertikalen Rollbalkens
sbPageDown	7	Unterer Rollbereich eines vertikalen Rollbalkens
sbIndicator	8	Zeichen für Positionsindikator

Bitmasken für TWindow.Flag

Konstante	Wert	Bedeutung
wfMove	$01	Obere Linie des Rahmens kann mit der Maus erfaßt und das Fenster bewegt werden
wfGrow	$02	Rahmen hat ein Größenänderungsfeld
wfClose	$04	Rahmen hat ein Schließfeld
wfZoom	$08	Rahmen hat ein Zoomfeld

Farbpalettenkonstanten für Fenster

Konstante	Wert	Bedeutung
wpBlueWindow	0	Gelbe Zeichen vor blauem Hintergrund
wpCyanWindow	1	Blaue Zeichen vor türkisfarbenem Hintergrund
wpGrayWindow	2	Schwarze Zeichen vor grauem Hintergrund

Standardbefehle

Konstante	Wert	Bedeutung
cmReceivedFocus	50	Objekt hat den Fokus übernommen
cmReleasedFocus	51	Objekt hat den Fokus abgegeben
cmCommandSetChanged	52	Verfügbarer Befehlssatz hat sich geändert
cmScrollBarChanged	53	Rollbalkenwerte haben sich geändert
cmScrollBarClicked	54	Rollbalken wurde angeklickt
cmSelectWindowNum	55	Fenster über seine Nummer selektieren
cmRecordHistory	56	Daten im Eingabefeld in Eingabeaufzeichnungsliste speichern

Variablen

Variable	Typ	Anfangswert	Bedeutung
ErrorAttr	Byte	$CF	Wird von *TView.GetColor* im Fehlerfall zurückgegeben
MinWinSize	*TPoint*	(X:16; Y:6)	Minimalgröße eines Fensters
ShadowSize	*TPoint*	(X:2; Y:1)	Größe des Schattens bei Fenstern
ShadowAttr	Byte	$08	Bildschirmattribut für Schatten

Routinen

Prozedur	Operation
RegisterViews	Registriert alle Objekte dieser Unit für die Arbeit mit einem Stream

Funktion	Operation
Message	Sendet benutzerdefinierte Botschaften an Objekte

Unit Dialogs

Diese Unit stellt Eingabeelemente bereit, die in einem Dialogfenster verwendet werden können. Dazu gehört diese spezielle Fensterart selbst, aber auch Aktionsschalter, Auswahl- und Eingabefelder, Eingabeaufzeichnungslisten und Objekte zur Beschriftung.

Typen

Typ	Funktion
TButton	Aktionsschalter, die Befehle absetzen
TCheckBoxes	Markierungsfelder
TCluster	Abstrakter Objekttyp für Auswahlfelder
TDialog	Spezialisierter Objekttyp für Dialogfenster
THistory	Eingabeaufzeichnungslisten
TInputLine	Eingabefeld
TLabel	Beschriftung
TListBox	Liste, aus der der Benutzer Einträge auswählen kann
TParamText	Text mit Formatierungsanweisungen ausgeben

Typ	*Funktion*
TRadioButtons	Schaltfelder
TSItem	Speichert Strings in einer verketteten Liste; benutzt von *TCluster*-Objekten
TStaticText	Objekt zur Ausgabe von Text

Konstanten

Konstanten für TButton

Konstante	Wert	Bedeutung
bfNormal	$00	Normaler Schalter
bfDefault	$01	Voreingestellter Schalter
bfLeftJust	$02	Schalterbeschriftung linksbündig ausgeben
bfBroadCast	$04	Schalter sendet über *Message* eine *evBroadcast*-Botschaft statt über *PutEvent* ein *evCommand*

Prozeduren und Funktionen

Funktion	Operation
NewSItem	Fügt einer Liste einen neuen String-Eintrag hinzu

Prozedur	Operation
RegisterDialogs	Registriert alle Objekte dieser Unit für die Arbeit mit einem Stream

Unit App

Diese Unit, zu der auch der Sourcecode mitgeliefert wird, stellt die notwendigen Elemente zur Entwicklung von Turbo-Vision-Programmen zur Verfügung. Vier wichtige Objekttypen sind in dieser Unit definiert. Die Objekte *TProgram* und *TApplication* liefern die Grundlage für Ihre Programme und *TBackGround* den Hintergrund, vor dem die anderen Objekte des Programms erscheinen. Die Aktionen dieser Objekte werden koordiniert von dem Objekt *TDeskTop*.

Typen

Typ	Funktion
TApplication	Objekt, das Bildschirm, Speicher und Ereignisse verwaltet, Fehler abfängt usw., die Grundlage Ihrer Programme
TBackGround	Gerasterter Hintergrund für die Arbeitsfläche
TDeskTop	Gruppe der Bildschirm- und Dialogfenster eingefügt werden
TProgram	Abstrakter Objekttyp, Vorfahre von *TApplication*

Variablen

Variable	Typ	Anfangswert	Bedeutung
Application	*PProgram*	**nil**	Zeiger auf eine Instanz von *TApplication*
DeskTop	*PDeskTop*	**nil**	Zeiger auf eine Instanz von *TDeskTop*
StatusLine	*PStatusLine*	**nil**	Zeiger auf eine Instanz von *TStatusLine*
MenuBar	*PMenuView*	**nil**	Zeiger auf eine Instanz von *TMenuView*

Prozedur

Prozedur	Operation
RegisterApp	Registriert alle Objekte dieser Unit für die Arbeit mit einem Stream

Unit Menus

Diese Unit stellt Objekte und Routinen zum Aufbau eines Menüsystems in Turbo-Vision-Programmen bereit. Zur Verfügung stehen sowohl Pull-Down-Menüs und Menüleisten, als auch Statuszeilen.

Typen

Typ	Funktion
TMenu	Verkettete Liste von *TMenuItem*-Records
TMenuBar	Menüleiste
TMenuBox	Pull-Down- oder Pop-Up-Menüs

Typ	Funktion
TMenuItem	Record, der Beschriftung, Tastenkürzel, Befehl und Hilfenummer miteinander verknüpft; definiert eine Menüoption
TMenuStr	Stringtyp für Menüoptionen
TMenuView	Abstrakter Objekttyp, von dem Menüleisten und -fenster abgeleitet sind
TStatusDef	Record zur Verknüpfung eines Bereichs von Hilfenummern mit Einträgen in der Statuszeile
TStatusItem	Record, der Beschriftung, Tastenkürzel und Befehl miteinander verknüpft; definiert eine Option in der Statuszeile
TStatusLine	Objekt zur Ausgabe der Statuszeile

Prozeduren und Funktionen

TMenuItem-Funktionen

Funktion	Operation
NewItem	Erstellt einen Menüeintrag
NewLine	Erstellt eine Linie zwischen Menüoptionen
NewSubMenu	Erstellt ein Untermenü, das mit der Menüleiste oder einem Menü verbunden ist

TMenu-Routinen

Routine	Operation
NewMenu-Funktion	Reserviert Speicher für ein Menü
DisposeMenu-Prozedur	Gibt den vom Menü belegten Speicher frei

TStatusLine-Funktionen

Funktion	Operation
NewStatusDef	Definiert einen Bereich von Hilfenummern und einen Zeiger auf eine Liste von Statuszeileneinträgen
NewStatusKey	Definiert einen Statuszeileneintrag und verknüpft mit ihm einen Befehl und optional ein Tastenkürzel

Registrierungsprozedur

Prozedur	Operation
RegisterMenus	Registriert alle Objekte dieser Unit für die Arbeit mit einem Stream

Unit Drivers

Diese Unit liefert Maus- und Tastaturunterstüzung, Routinen zur Ermittlung des Bildschirmadapters, zur Fehlerbehandlung und zur Verwaltung des Speichers.

Typen

Typ	Funktion
TEvent	Recordtyp zur Speicherung eines Ereignisses
TSysErrorFunc	Funktionstyp der Fehlerabfangroutine

Konstanten

Konstanten zur Abfrage der Maus

Konstante	Wert	Bedeutung
mbLeftButton	$01	Linker Mausknopf
mbRightutton	$02	Rechter Mausknopf

Bitmasken zur Ermittlung der Art eines Ereignisses

Konstante	Wert	Bedeutung
evMouseDown	$0001	Mausknopf gedrückt
evMouseUp	$0002	Mausknopf losgelassen
evMouseMove	$0004	Position des Mauszeiger geändert
evMouseAuto	$0008	Automatisch wiederholtes Mausereignis
evKeyDown	$0010	Taste gedrückt
evCommand	$0100	Befehlsereignis
evVBroadcast	$0200	Rundruf-Ereignis

Bitmasken zur Ermittlung der Kategorie eines Ereignisses

Konstante	Wert	Bedeutung
evNothing	$000	Abgearbeitetes Ereignis
evKeyboard	$0010	Tastaturereignis
evMouse	$000F	Mausereignis
evMessage	$FF00	Botschaft oder Befehlsereignis

Konstanten zur Ermittlung des Tastaturstatus

Konstante	Wert	Bedeutung
kbRightShift	$0001;	Rechte Shift-Taste gedrückt
kbLeftShift	$0002;	Linke Shift-Taste gedrückt
kbCtrlShift	$0004;	Crtl-Taste gedrückt
kbAltShift	$0008;	Alt-Taste gedrückt
kbScrollState	$0010;	Scroll-Lock aktiviert
kbNumState	$0020;	Num-Lock aktiviert
kbCapsState	$0040;	Caps-Lock aktiviert
kbInsState	$0080;	Insert aktiviert

Standardbefehle

Konstante	Wert	Bedeutung
cmValid	0	Gültigkeit eines neuen *View*-Objekts überprüfen
cmQuit	1	Programm beenden
cmError	2	Nicht definiert
cmMenu	3	Menüleiste aktivieren
cmClose	4	Aktuelles Fenster schließen
cmZoom	5	Fenster zoomen oder alte Größe wiederherstellen
cmResize	6	Größe des Fensters ändern
cmNext	7	Nächstes Fenster selektieren
cmPrev	8	Voriges Fenster selektieren

Standardbefehle für TDialog

Konstante	Wert	Bedeutung
cmOK	10	»Ok«-Schalter aktiviert
cmCancel	11	»Cancel«-Schalter aktiviert oder [Esc] gedrückt
cmYes	12	»Yes«-Schalter aktiviert
cmNo	13	»No«-Schalter aktiviert
cmDefault	14	Voreingestellter Schalter aktiviert oder [Enter] gedrückt

Bildschirmmodus

Konstante	Wert	Bedeutung
smBW80	$0002	Schwarzweißer Modus
smCO80	$0003	Farbmodus
smMono	$0007	Monochromer Modus
smFont8x8	$0100	43- oder 50-Zeilen-Modus (EGA/VGA)

Variablen

Initialisierte Variablen

Variable	Typ	Anfangswert	Bedeutung
ButtonCount	Byte	0	Zahl der Mausknöpfe
MouseEvents	Boolean	False	Gibt an, ob eine Maus ermittelt wurde
MouseReverse	Boolean	False	Maus für Linkshänder, wenn True
DoubleDelay	Word	8	Maximaler Zeitraum zwischen den Klicks eines Doppelklicks
RepeatDelay	Word	8	Zeitraum, bis sich wiederholende Mausereignisse generiert werden

Nicht initialisierte Variablen

Variable	Typ	Bedeutung
MouseIntFlag	Byte	Nur für internen Gebrauch
MouseButtons	Byte	Welcher Knopf gedrückt
MouseWhere	*TPoint*	Position des Mauszeigers
StartUpMode	Word	Bildschirmmodus beim Programmstart
ScreenMode	Word	Aktueller Bildschirmmodus
ScreenWidth	Byte	Breite des Bildschirms in Spalten
ScreenHeight	Byte	Höhe des Bildschirms in Zeilen
CheckSnow	Boolean	Ausgabe verzögern, um Schnee auf dem Bildschirm zu vermeiden (CGA)
HiResScreen	Boolean	43- oder 50-Zeilen-Modus möglich (EGA/VGA)
ScreenBuffer	Pointer	Zeiger auf den Bildschirmspeicher
CursorLines	Word	Speichert den Wert für die Cursorform

Variablen zur Fehlerbehandlung

Variable	Typ	Anfangswert	Bedeutung
SysErrorFunc	*TSysErrorFunc*	*SystemError*	Diese Funktion wird aufgerufen, wenn ein kritischer Fehler auftritt
SysColorAtr	Word	$4E4F	Bildschirmattribute zur Ausgabe von Fehlermeldungen auf Farbbildschirmen
SysMonoAttr	Word	$7070	Bildschirmattribute zur Ausgabe von Fehlermeldungen auf Monochrombildschirmen

Variable	Typ	Anfangswert	Bedeutung
CtrlBreakHit	*Boolean*	*False*	Zeigt an, ob der Benutzer *Ctrl-Break* gedrückt hat
SaveCtrlBreak	Boolean	False	Speichert den Status der *Ctrl-Break*-Überprüfung am Programmbeginn

Prozeduren und Funktionen

Prozeduren zur Verwaltung von Ereignissen

Prozedur	Operation
InitEvents	Initialisiert die Verwaltung von Ereignissen
DoneEvents	Beendet die Ereignis-Verwaltung
ShowMouse	Gibt den Mauszeiger aus
HideMouse	Schaltet den Mauszeiger aus
GetMouseEvent	Erstellt einen Ereignis-Record, der ein Mausereignis speichert
GetKeyEvent	Erstellt einen Ereignis-Record, der ein Tastaturereignis speichert

Bildschirm-bezogene Prozeduren

Prozedur	Operation
InitVideo	Initialisiert die Verwaltung des Bildschirms
DoneVideo	Restauriert den beim Programmstart vorgefundenen Zustand
SetVideoMode	Setzt einen Bildschirmmodus
ClearScreen	Löscht den Bildschirm in jedem Modus

Fehlerbehandlungsfunktion

Funktion	Operation
SystemError	Gibt eine Fehlermeldung in der letzten Zeile des Bildschirms aus und fragt an, ob die fehlerauslösende Operation wiederholt oder abgebrochen werden soll

Prozeduren zur Fehlerbehandlung

Prozedur	Operation
InitSysError	Übernimmt die zum Abfangen von Fehlern relevanten Interruptvektoren
DoneSysError	Restauriert die übernommenen Interruptvektoren

Tastaturbezogene Funktionen

Funktion	Operation
GetAltChar	Konvertiert Tastaturcode in Zeichen
GetAltCode	Konvertiert Zeichen in Tastaturcode

String-Formatierungsprozedur

Prozedur	Operation
FormatStr	Formatiert einen String, indem die übergebenen Parameter eingefügt werden

Puffer-Prozeduren

Prozedur	Operation
MoveBuf	Kopiert den Inhalt eines Puffers in einen anderen
MoveChar	Kopiert ein Zeichen ein- oder mehrmals in den Puffer
MoveCStr	Kopiert einen Tilde-zeichen enthaltenden String in den Puffer
MoveStr	Kopiert einen String in den Puffer

Stringlängen-Funktion

Funktion	Operation
CStrLen	Gibt die Länge eines Strings zurück, wobei darin enthaltene Tilde-Zeichen nicht beachtet werden

String-Formatierungsprozedur

Prozedur	Operation
InitDrivers	Initialisierung der Unit *Drivers*

Unit TextView

Diese Unit stellt einige Objekte bereit, die darauf spezialisiert sind, Text innerhalb eines Fensters auszugeben, wobei der Text innerhalb des Fensters in zwei Richtungen gerollt werden kann.

Typen

Typ	Funktion
TTerminal	Fernschreiber-ähnliche Text-Ein-/Ausgabeeinheit
TTerminalBuffer	Ringpuffer, benutzt von *TTerminal*
TTextDevice	Abstraktes Objekt zur Realisierung einer Text-Ein-/Ausgabe-einheit

Prozedur

Prozedur	Operation
AssignDevice	Ordnet eine Textdatei dem Bildschirm als Ein-/Ausgabe-Einheit zu

Unit Memory

Diese Unit implementiert die Speicherverwaltung von Turbo Vision. Damit ist es möglich, Programme zu schreiben, die ihren Speicherbedarf überwachen und daher der Gefahr, wegen Speichermangels abzubrechen, vorbeugen können.

Variable

Variable	Typ	Anfangswert	Bedeutung
LowMemSize	Word	4096 **div** 16	Größe des Sicherheitsbereichs

Prozeduren und Funktionen

Prozedur	Operation
InitMemory	Initialisiert die Speicherverwaltung
DoneMemory	Gibt eventuell belegten EMS-Speicher wieder frei
GetBufMem	Reserviert Speicher für den Puffer einer Gruppe
FreeBufMem	Gibt den für den Puffer einer Gruppe belegten Speicher frei

Funktion	Operation
LowMemory	Gibt an, ob der Sicherheitsbereich in eine Speicherreservierung einbezogen wurde
MemAlloc	Reserviert Speicher, wobei der Sicherheitsbereich beachtet wird

Unit HistList

Diese Unit enthält alle Variablen, Prozeduren und Funktion, die notwendig sind, um Eingabeaufzeichnungslisten bereitzustellen und zu verwalten.

Variablen

Variable	Typ	Anfangswert	Bedeutung
HistoryBlock	Pointer	**nil**	Zeigt auf den Puffer zur Speicherung der Eingabeaufzeichnungslisten
HistorySize	Word	1024	Größe des Puffers für Eingabeaufzeichnungslisten
HistoryUsed	Word	0	Zeigt die Größe des benutzten Bereichs im Puffer zur Speicherung der Eingabeaufzeichnungslisten

Prozeduren und Funktionen

Prozedur	Operation
HistoryAdd	Speichert einen Eintrag in einer Eingabeaufzeichnungsliste
ClearHistory	Löscht alle Eingabeaufzeichnungslisten
InitHistory	Initialisiert die Verwaltung von Eingabeaufzeichnungslisten
DoneHistory	Löscht die Listen und gibt den belegten Speicher frei

Funktion	Operation
HistoryCount	Gibt die Zahl der Strings in der Eingabeaufzeichnungsliste zurück
HistoryStr	Gibt einen bestimmten String aus der Eingabeaufzeichnungsliste zurück

14. Verzeichnis der Objekte

Dieses Kapitel behandelt in alphabetischer Folge die in Turbo Vision definierten Objekttypen. Jeder Eintrag beginnt mit einer Beschreibung der Funktion und Verwendungsweise des Objekts. Danach werden alle Felder und Methoden und – wenn es sich um ein *View*-Objekt handelt – die verwendete Farbpalette beschrieben.

Wollen Sie sich über ein bestimmtes Objekt informieren, müssen Sie immer beachten, daß dieses Objekt viele Eigenschaften von seinen Vorfahren geerbt haben kann. Bei der Beschreibung eines Objekts haben wir uns aber darauf beschränkt, nur die jeweils *neuen* Felder und Methoden, darin eingeschlossen natürlich auch jene, die geerbte Methoden überschreiben, darzustellen.

Nehmen wir an, Sie wollten wissen, welche Funktion das Feld *Owner* eines *TLabel*-Objekts eigentlich hat. Sie sehen zunächst in dem entsprechneden Eintrag nach, finden aber dort das Feld *Owner* nicht. Das Objekt *TLabel* ist ein Nachkomme von *TStaticText*. Aber auch hier ist das Feld *Owner* nicht beschrieben. Der unmittelbare Vorfahre von *TStaticText* ist das Objekt *TView*. Und hier endlich finden Sie *Owner* erklärt. Um Ihnen den Überblick über die Abstammung eines Objekts zu erleichtern, zeigt eine kleine Grafik bei jedem Objekt dessen Platz in der Hierarchie der von Turbo Vision definierten Objekte.

Jeder Eintrag in diesem Verzeichnis entspricht im wesentlichen dem folgenden Beispiel:

Objektname **UNIT-NAME**

Rechts neben dem Namen des Objekts ist die Unit angegeben, die Sie einbinden müssen, wenn Sie dieses Objekt

verwenden wollen. Die Grafik zeigt den Platz des Objekts innerhalb der Hierarchie der in Turbo Vision definierten Objekte.

Felder Dieser Abschnitt führt alle Felder auf, die dieses Objekt neu definiert.

FeldName `FeldName: Feldtyp;`

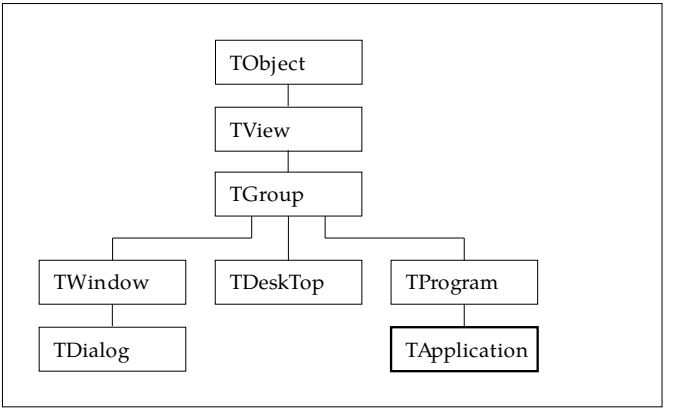
Hier folgt der oben angeführten formalen Deklaration eine Beschreibung der Funktion dieses Feldes, seiner Verwendung usw.

Methoden Dieser Abschnitt führt die Methoden auf, die das Objekt entweder neu definiert, oder die die von seinen Vorfahren geerbten Methoden überschreiben. Wenn vorhanden, beginnt die Liste mit den Konstruktor- und den Destruktor-Methoden.

Name `procedure Name(var S: String);`

Hier folgen der oben angeführten formalen Deklaration eine Beschreibung der Methode und Querverweise auf damit zusammenhängende oder ähnliche Felder und Methoden anderer Objekte.

TApplication APP



TApplication bildet zusammen mit seinem Vorfahren *TProgram* das Fundament aller Turbo-Vision-Programme, da alle diese Programme von einem dieser beiden Objekte abgeleitet werden müssen. *TApplication* unterscheidet sich von *TProgram* nur durch seine *Init*- und seine *Done*-Methode. Auf welchem dieser beiden Objekte Sie Ihr Programm aufbauen,

hängt von Ihrer Anwendung ab. In der Regel wird aber ein Turbo-Vision-Programm von *TApplication* abgeleitet.

TApplication ist als Nachkomme von *TGroup* definiert, da diesem Objekt Instanzen von *TDeskTop*, *TStatusLine* und *TMenuBar* eingefügt werden müssen.

Methoden

Init

```
constructor Init;
```

Diese Methode ruft *InitMemory*, *InitMouse*, *InitVideo*, *InitEvents*, *InitSysError* und schließlich die *Init*-Methode von *TGroup* auf, die ihrerseits die Gruppe *TApplication* initialisiert und ihr den ganzen Bildschirm als Bereich zuordnet. Im Feld *Status* werden die Bits *sfVisible*, *sfSelected*, *sfFocused*, *sfModal* und *sfExposed* gesetzt. Zur schnelleren Darstellung wird ein Bildschirmpuffer angelegt. Darauf ruft *TApplication.Init* die Methoden *InitDeskTop*, *InitStatusLine* und *InitMenuBar* auf. Diese drei dadurch initialisierten Objekte werden anschließend in die Gruppe *TApplication* eingefügt. Siehe auch: *TGroup.Init*, *TProgram.Init*, *TProgram.InitDeskTop*, *TProgram.InitStatusLine*, *TProgram.InitMenuBar*.

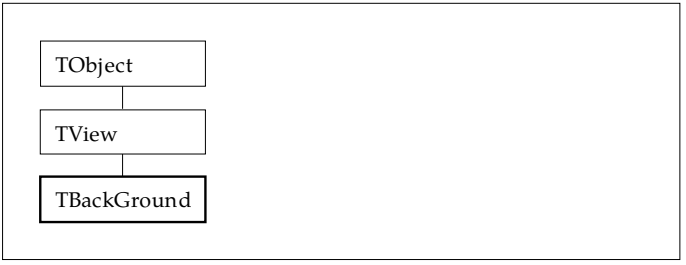
Done

```
destructor Done; virtual;
```

Ruft *TProgram.Done* auf, restauriert Interrupt-Vektoren und den Bildschirm-Modus und löscht schließlich den Bildschirm. Jedes Turbo-Vision-Programm muß diesen Destruktor aufrufen, um ein sicheres und sauberes Beenden des Programms zu gewährleisten. Diese Methode ist folgendermaßen implementiert:

```
destructor TApplication.Done;
begin
  TProgram.Done;
  DoneHistory;
  DoneSysError;
  DoneEvents;
  DoneVideo;
  DoneMemory;
end;
```

TBackGround APP



TBackGround ist ein *View*-Objekt, das lediglich ein einheitlich gemustertes Rechteck darstellt. Üblicherweise gehört es zu einer Instanz von *TDeskTop*.

Felder

Pattern

```
Pattern: Char;
```

Das Zeichen, mit dem der Hintergrund gefüllt wird.

Methoden

Init

```
constructor Init(var Bounds: TRect; APattern: Char);
```

Initialisiert über einen Aufruf von *TView.Init*(*Bounds*) ein *TBackGround*-Objekt für den in *Bounds* angegebenen Bereich. Im Feld *GrowMode* werden die Bits *gfGrowHiX* und *gfGrowHiY* gesetzt; das Feld *Pattern* erhält den Wert *APattern*. Siehe auch: *TView.Init*, *TView.GrowMode*, *Pattern*.

Load

```
constructor Load(var S: TStream);
```

Kreiert ein *TBackGround*-Objekt und liest die zur Initialisierung notwendigen Werte aus dem *TStream*. Dazu ruft diese Methode zunächst *TView.Load*(*S*) und dann *S.Read*(*Pattern*,*SizeOf*(*Pattern*)) auf. Siehe auch: *TView.Load*, *TStream.Read*.

Draw

```
procedure Draw; virtual;
```

Füllt das Rechteck, für das dieses Objekt verantwortlich ist, mit den Zeichen *Pattern*. Die verwendete Farbe wird über den Paletteneintrag 1 ermittelt.

GetPalette

```
function GetPalette: PPalette; virtual;
```

Gibt einen Zeiger auf die voreingestellte Farbpalette *CBackGround* zurück.

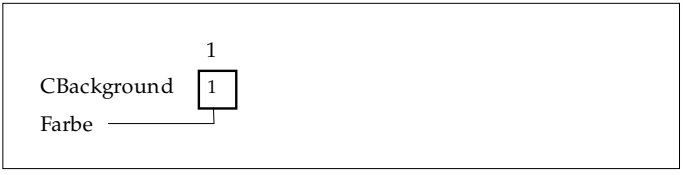
Store

```
procedure Store(var S: TStream);
```

Speichert ein *TBackGround*-Objekt in den *TStream*. Dazu ruft diese Methode zunächst *TView.Store*(*S*) und dann *S.Write*(*Pat-*

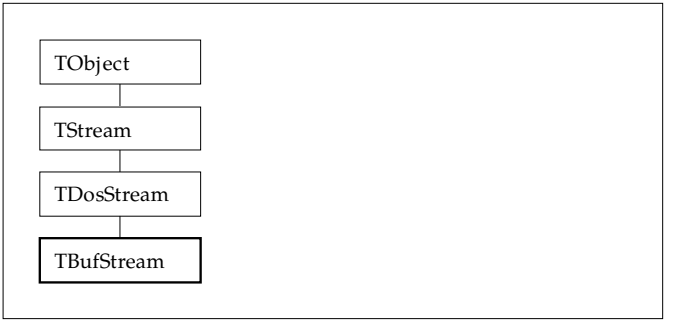
tern,SizeOf(Pattern)) auf. Siehe auch: *TView.Store*, *TStream.Write*, *TBackGround.Load*.

Farbpalette Dieses Objekt greift über die Palette *CBackGround* auf den ersten Eintrag der aktuellen Farbpalette des Programms zu.



TBufStream

OBJECTS



TBufStream ist ein Nachfolger von *TDosStream*, der im Unterschied zu diesem Ein- und Ausgaben über einen Puffer abwickelt. Die zusätzlichen Felder enthalten einen Zeiger auf diesen Puffer und geben dessen Größe an. Zwei weitere Felder speichern die aktuelle Position in diesem Puffer und zeigen seinen Füllungsgrad an. Zusätzlich zu den acht Methoden, die auch schon *TDosStream* überschreibt, definiert *TBufStream* die Methode *Flush* neu. Der Konstruktor *Init* dieses Objekts öffnet über einen Aufruf von *TDosStream.Init* eine Datei und reserviert darüber hinaus mittels *GetMem* einen Puffer.

Felder

Buffer `Buffer: Pointer;`

Zeiger auf den Anfang des dem Stream zugeordneten Puffers.

BufSize `BufSize: Word;`

Größe des Puffers in Bytes.

BufPtr `BufPtr: Word;`

Ein Offset auf *Buffer*; gibt dadurch die aktuelle Position im Puffer wieder.

BufEnd `BufEnd: Word;`

Sofern der Puffer nicht schon gefüllt ist, zeigt dieser Offset auf *Buffer* auf das letzte genutzte Byte im Puffer.

Methoden

Init `constructor Init(FileName: FNameStr; Mode, Size: Word);`

Über einen Aufruf von *TDosStream.Init* wird in Abhängigkeit von dem Parameter *Mode* eine Datei unter dem Namen *FileName* neu angelegt oder eine solche, schon existierende Datei geöffnet. Darauf reserviert *Init* mittels *GetMem* einen Speicherbereich von *Size* Größe für den Puffer und legt die entsprechenden Werte in den Feldern *Buffer* und *BufSize* ab. Siehe auch: *TDosStream.Init*.

Done `destructor Done; virtual;`

Erzwingt über *Flush*, daß, sofern notwendig, der Pufferinhalt in die Datei geschrieben wird, schließt diese Datei und gibt den von dem Stream und dem Puffer belegten Speicher frei. Siehe auch: *TBufStream.Flush*.

Flush `procedure Flush; virtual;`

Enthält *ErrorInfo* den Wert *stOK*, schreibt diese Methode den Pufferinhalt in die Datei, sofern dies notwendig ist. Die Felder *BufPtr* und *BufEnd* werden auf 0 gesetzt.

GetPos `function GetPos: LongInt; virtual;`

Gibt die aktuelle Position im Stream zurück. (Nicht zu verwechseln mit der aktuellen Position im Puffer, die in *BufPtr* abgelegt ist.) Siehe auch: *TStream.Get*, *TStream.Put*.

GetSize `function GetSize: LongInt; virtual;`

Erzwingt über *Flush*, daß, sofern notwendig, der Pufferinhalt in die Datei geschrieben wird, und gibt darauf die Größe des Stream in Bytes zurück. Siehe auch: *TStream.CopyFrom*.

Read `procedure Read(var Buf; Count: Word); virtual;`

Enthält *ErrorInfo* den Wert *stOK*, liest diese Methode, beginnend an der aktuellen Position im Stream, *Count* Bytes aus dem Stream und speichert sie in *Buf*. Siehe auch: *stReadError*, *TBufStream.GetPos*, *TBufStream.Seek*.

Seek `procedure Seek(Pos: LongInt); virtual;`

Erzwingt über *Flush*, daß, sofern notwendig, der Pufferinhalt

in die Datei geschrieben wird, und setzt dann die aktuelle Position auf die Stelle *Pos* Bytes ab Beginn des Stream. Der Stream beginnt mit der Position 0. Siehe auch: *TBufStream.GetPos*, *TStream.Put*, *TStream.Get*, *TStream.CopyFrom*.

Truncate

```
procedure Truncate; virtual;
```

Erzwingt über *Flush*, daß, sofern notwendig, der Pufferinhalt in die Datei geschrieben wird, und löscht dann ab der aktuellen Position bis zum Ende des Stream alle Daten. Die Position selbst verändert sich nicht, zeigt also nach der Operation auf das Ende des Stream. Siehe auch: *TBufStream.GetPos*, *TBufStream.Seek*.

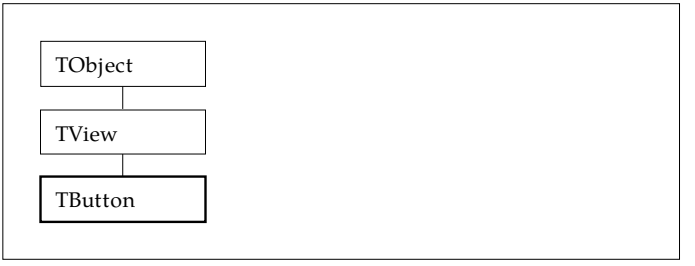
Write

```
procedure Write(var Buf; Count: Word); virtual;
```

Enthält *ErrorInfo* den Wert *stOK*, schreibt diese Methode, beginnend an der aktuellen Position im Stream, *Count* Bytes aus *Buf* in den Stream. Siehe auch: *TBufStream.Read*, *TDosStream.GetPos*, *TDosStream.Seek*.

TButton

DIALOGS



TButton stellt ein spezielles Eingabemedium zur Verfügung: einen Schalter, der einen Befehl absetzt, sobald er »gedrückt« wird. Ein solches Element stellt sich auf einem Farbbildschirm als beschriftetes Rechteck dar, unterhalb dessen ein Schatten sichtbar ist. Dieser Schatten verschwindet, sobald der Schalter betätigt wird: Es scheint, als werde ein Schaltknopf gedrückt. Bei monochromen Bildschirmen werden Schalter in eckigen Klammern angezeigt und die Selektion eines Schalters wird durch andere ASCII-Zeichen gekennzeichnet. Über die Beschriftung kann auch ein Tastenkürzel definiert werden, indem ein Buchstabe dieses Titels von Tilde-Zeichen (~) umgeben wird. Dadurch ist es möglich, den Schalter mittels [Alt] und der Taste, die diesem Zeichen zugeordnet ist, zu aktivieren. Das gleiche erreichen Sie, wenn Sie den Schalter mit der Maus anklicken, oder ihn mit [Tab] und [Leertaste] selektieren und dann [Enter] drücken.

In der integrierten Entwicklungsumgebung werden diese Schalter beispielsweise dazu verwendet, einen modalen Dialog zu beenden, wenn der Benutzer die mit »OK« oder »Cancel« beschrifteten Elemente »drückt«. Eine unterscheidende Farbgebung zeigt an, welcher von mehreren Schaltern gerade selektiert ist oder welche Schalter deaktiviert sind.

Felder

Title

```
Title: PString;
```

Ein Zeiger auf den String, der zur Beschriftung des Schalters dient.

Command

```
Command: Word;
```

Der Befehl, der abgesetzt wird, sobald der Benutzer den Schalter betätigt. Siehe auch: *TButton.Init*, *TButton.Load*.

Flags

```
Flags: Byte;
```

Die in diesem Byte gesetzten oder gelöschten Bits legen fest, ob dieser Schalter der aktuelle ist, ob die Beschriftung linksbündig oder zentriert ausgegeben wird usw. Siehe auch: *TButton.Draw*. Die möglichen Werte für *Flags* sind in Kapitel 15, »Außerhalb der Objekte«, unter dem Stichwort »*bfXXX-Konstanten*« erläutert.

AmDefault

```
AmDefault: Boolean;
```

Wenn True, ist dieser Aktionsschalter voreingestellt und kann daher über [Enter] selektiert werden.

Methoden

Init

```
constructor Init(var Bounds: TRect; ATitle: TTitleStr;
                 ACommand: Word; AFlags: Byte);
```

Initialisiert über *TView.Init*(*Bounds*) ein *TButton*-Objekt mit der in *Bounds* angegebenen Größe. Die Felder *Title*, *Command* und *Flags* erhalten die in *ATitle*, *ACommand* und *AFlags* übergebenen Werte. Der Parameter *AFlags* dient zwei Zwecken: Hat er das Bit *bfDefault* gesetzt, wird in *AmDefault* True eingetragen. Darüber hinaus gibt *AFlags* über das Bit *bfLeftJust* an, ob die Beschriftung des Schalters linksbündig oder zentriert auszugeben ist. Im Feld *Options* werden die Bits *ofSelectable*, *ofFirstClick*, *ofPostProcess* und *ofPreProcess*, im Feld *EventMask* das Bit *evBroadCast* gesetzt. Wenn der Befehl *ACommand* nicht aktivierbar ist (was die Anweisung *CommandEnabled*(*ACommand*) überprüft), erhält das Feld *State* den Wert *sfDisabled*. Siehe auch: *TView.CommandEnabled*, *TView.Init*.

Load

```
constructor Load(var S: TStream);
```

Erstellt ein *TButton*-Objekt und initialisiert es mit Werten, die aus dem Stream mittels *TView.Load(S)*, dem ein Aufruf von *S.Write* folgt, eingelesen werden. Weiterhin wird überprüft, ob *Command* aktivierbar ist oder nicht, und das Feld *State* entsprechend gesetzt (vergleiche *TButton.Init*). Diese Methode wird zusammen mit *TButton.Store* genutzt, um Objekte des Typs *TButton* in einem Stream zu speichern bzw. wieder zu laden. Siehe auch: *TView.Load*, *TButton.Store*.

Done `destructor Done; virtual;`

Gibt den von *Title* belegten Speicher frei und ruft dann *TView.Done* auf, um das Objekt vom Bildschirm zu entfernen. Siehe auch: *TView.Done*

Draw `procedure Draw; virtual;`

Gibt das Objekt in den Farben aus, die dem Status des Schalters entsprechen, und versieht es in der Art, wie es das Bit *bfLeftJust* in *Flags* festlegt, mit der Beschriftung *Title*^.

GetPalette `function GetPalette: PPalette; virtual;`

Diese Methode gibt einen Zeiger auf die Farbpalette *CButton* zurück.

HandleEvent `procedure HandleEvent(var Event: TEvent); virtual;`

Diese Methode verbindet den Befehl *Command* mit dem Schalter. Drückt der Benutzer den Schalter, wird *Command* im gleichnamigen Feld von *Event* eingetragen, das Feld *Event.InfoPtr* auf *@Self* gesetzt und der so belegte *Event*-Record mittels eines Aufrufs der Methode *PutEvent* den der Gruppe angehörenden Objekten übermittelt. Siehe auch: *TView.HandleEvent*.

MakeDefault `procedure MakeDefault(Enable: Boolean);`

Ist der Schalter bereits der aktuelle, tut diese Methode überhaupt nichts. Andernfalls wird der Gruppe, zu der das Objekt gehört, darüber informiert, daß dieser Schalter zu aktualisieren ist und ihr der Befehl *cmGrabDefault* (wenn *Enable* True ist) bzw. *cmReleaseDefault* (wenn *Enable* False ist) übergeben. Darauf wird der Schalter neu dargestellt, damit seine Statusänderung auch am Bildschirm sichtbar wird. Siehe auch: *bfDefault*, *TButton.AmDefault*.

Press `procedure Press; virtual;`

Sofern nicht überschrieben, sendet diese Methode ein Ereignis des Typs *evBroadcast*, das den Befehl *cmRecordHistory* übermittelt, an die Gruppe, zu der das Objekt gehört. Dadurch werden alle *THistory*-Objekte veranlaßt, die Eingabezeile des *TInputLine*-Objekts, mit dem sie verbunden

sind, zu speichern. Sie können diese Methode überschreiben, um eine andere Aktion auszulösen, wenn der Aktionsschalter »gedrückt« wird.

SetState

```
procedure SetState(AState: Word; Enable : Boolean);
virtual;
```

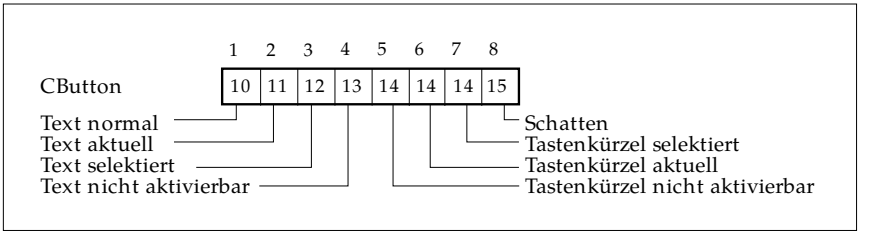
Ruft *TView.SetState*(*AState, Enabled*) und dann *TView.DrawView* auf, sofern das Bit *sfSelected* oder das Bit *sfActive* in *AState* gesetzt ist. Wenn *AState* den Wert *sfFocused* aufweist, also der Schalter fokussiert werden soll, wird über die Anweisung *MakeDefault*(*Enable*) veranlaßt, daß der bisher aktualisierte Schalter diesen Status an das diese Methode aufrufende *TButton*-Objekt abtritt. Siehe auch: *TView.SetState*, *TButton.Make Default*.

Store

```
procedure Store(var S: TStream);
```

Speichert über *TView.Store*(*S*) ein *TButton*-Objekt im angegebenen Stream. Die Methode *S.Write* übernimmt die Speicherung der Felder *Title* und *Command*. Zusammen mit *TButton.Load* erlaubt diese Methode die Ein- und Ausgabe eines *TButton*-Objekts über einen Stream. Siehe auch: *TView.Store*, *TButton.Load*, *TStream.Write*.

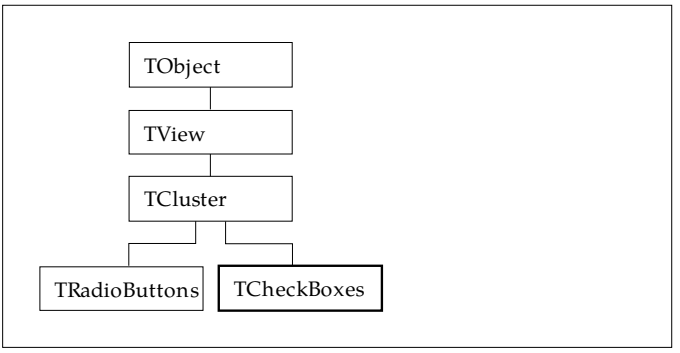
Farbpalette Dieses Objekt greift über die Palette *CButton* auf die Einträge 10 bis 15 der Palette *CDialog* zu.



TCheckBoxes

DIALOGS

TCheckBoxes ist, wie *TRadioButtons* auch, ein spezialisierter Nachkomme von *TCluster*. Jedoch kann in einem Markierungsfeld, das über *TCheckBoxes* erzeugt wird und bis zu 16 Schalter enthalten kann, jeder einzelne Schalter unabhängig von den anderen Schaltern »gedrückt« werden. Betätigt wird ein Schalter mittels der Maus, über die Tastenkombination [Alt][Buchstabe] oder auch über die Cursortasten. Der gerade aktuelle Schalter wird mittels [Leertaste] betätigt. Ist eine Option gewählt, erscheint ein X, ansonsten ist das entsprechende Feld leer. Markierungsfelder sind meist mit *TLabel-*



Objekten verbunden, wodurch die Anwahl eines solchen Schaltfeldes über die Selektion der mit ihm verbundenen Beschriftung möglich wird.

Felder

Über die von *TCluster* geerbten Felder hinaus definiert dieses Objekt keine weiteren Felder, interpretiert allerdings das Feld *Value* auf spezielle Weise: Ist ein Bit in *Value* gesetzt, gilt die damit verbundene Option als gewählt. Damit ist die Zahl der über eine Checkbox anwählbaren Optionen auf 16 beschränkt, wobei der ersten Option die Nummer 0 zugeordnet ist.

Methoden

Bemerkung: *TCheckBoxes* überschreibt weder den von *TCluster* ererbten Konstruktor oder Destruktor, noch die *HandleEvent*-Methode. Ihre abgeleiteten Objekte müssen dies – sofern überhaupt notwendig – selbst tun.

Draw

```
procedure Draw; virtual;
```

Stellt das *TCheckBoxes*-Objekt auf dem Bildschirm dar. Sofern nicht überschrieben, markiert diese Methode eine gewählte Option durch [X], ein nicht gewählte durch [ ]. Siehe auch: *TCluster.DrawBox*.

Mark

```
function Mark(Item: Integer): Boolean; virtual;
```

Gibt True zurück, wenn das Bit *Item* in *Value* gesetzt ist. In der Weise, wie *TCheckBoxes* das Feld *Value* interpretiert, bedeutet das, daß die Option *Item* ausgewählt ist. Wenn Ihr abgeleitetes Objekt *Value* anders interpretieren soll, müssen Sie diese Methode überschreiben. Siehe auch: *TCheckBoxes.Press*.

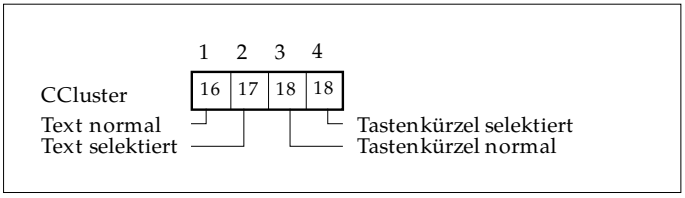
Press

```
procedure Press(Item: Integer); virtual;
```

Schaltet das Bit *Item* in *Value* um. In der Weise, wie *TCheckBoxes* das Feld *Value* interpretiert, bedeutet ein gesetztes Bit in *Value*, daß die entsprechende Option ausgewählt ist.

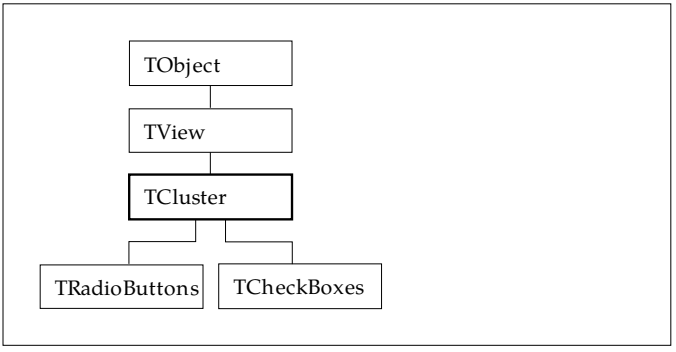
Wenn Ihr abgeleitetes Objekt *Value* anders interpretieren soll, müssen Sie diese Methode überschreiben. Siehe auch: *TCheckBoxes.Mark*.

Farbpalette Dieses Objekt greift über die Palette *CCluster* auf die Einträge 16 bis 18 der Palette *CDialog* zu.



TCluster

DIALOGS



Ein *TCluster* stellt ein Feld von Schaltern bereit, die alle auf die gleiche Weise reagieren. Dieser abstrakte Objekttyp bildet die gemeinsame Grundlage für die in Turbo Vision definierten Objekttypen *TRadioButtons* und *TCheckBoxes*. Solche Schaltfelder sind meist mit *TLabel*-Objekten verbunden, wodurch die Anwahl eines Schaltfelds über die Selektion der mit ihm verbundenen Beschriftung möglich wird.

Felder

Value `Value: Word;`

Dieses Feld speichert die aktuelle Einstellung der Optionen des Schaltfelds. Welche Bedeutung *Value* tatsächlich hat, wird festgelegt von den Methoden, die die von *TCluster* abgeleiteten Objekte definieren. Siehe auch: *TRadioButtons*, *TCheckBoxes*.

Sel `Sel: Integer;`

Bezieht sich auf den gerade selektierten Eintrag eines Schaltfelds.

Strings

```
Strings: TStringCollection;
```

Die Liste der Optionen des Schaltfeldes.

Methoden

Init

```
constructor Init(var Bounds: TRect; AStrings: PSItem);
```

Erstellt ein Schaltfeld in der in *Bounds* übergebenen Größe und versieht die einzelnen Schalter mit den in *AStrings* angegebenen Titeln. Dieser Parameter ist in der Regel ein verschachtelter Aufruf der Funktion *NewSItem*, da auf diese Weise ein ganzes Schaltfeld über einen einzigen Aufruf des Konstruktors initialisiert werden kann:

```
var
  Control: PView;

...
R.Assign(30, 5, 52, 7);
Control := New(PRadioButtons, Init(R;
  NewSItem('~V~orw/rts',
  NewSItem('~R~¡ckw/rts', nil))));
...
```

Siehe auch: *TSItem*, *NewSItem*.

Load

```
constructor Load(var S: TStream);
```

Erstellt ein *TCluster*-Objekt, das mit den über *TView.Load(S)* aus dem Stream gelesenen Werten initialisiert wird. Diese Methode wird zusammen mit *TCluster.Store* genutzt, um Objekte des Typs *TCluster* in einem Stream zu speichern, bzw. wieder zu laden. Siehe auch: *TCluster.Store*, *TStream.Read*, *TView.Load*.

Done

```
destructor Done; virtual;
```

Gibt den Speicher, den die zur Beschriftung des Schaltfelds dienenden Strings belegen, wieder frei und veranlaßt dann über einen Aufruf von *TView.Done*, daß das Schaltfeld vom Bildschirm entfernt wird. Siehe auch: *TView.Done*.

DataSize

```
function DataSize: Word; virtual;
```

Diese pseudo-abstrakte Methode gibt die Größe von *Value* zurück. Sie müssen diese Methode bei von *TCluster* abgeleiteten Objekten überschreiben, um zu gewährleisten, daß die Methoden *GetData* und *SetData* korrekt arbeiten. Siehe auch: *TCluster.GetData*, *TCluster.SetData*.

DrawBox

```
procedure DrawBox(Icon: String; Marker: Char);
```

Wird von den *Draw*-Methoden abgeleiteter Objekttypen aufgerufen, und erledigt die Ausgabe der Markierungskästchen für jeden Eintrag des Schaltfeldes. Der Parameter *Icon* ist ein fünf Zeichen langer String (' [ ] ' für Markierungsfelder und ' ( ) ' für Schaltfelder). Das Zeichen *Marker* ('X' für Markierungsfelder und '.' für Schaltfelder) wird benutzt, um anzuzeigen, daß eine Option ausgewählt wurde. Siehe auch: *TCheckBoxes.Draw*, *TRadioButtons.Draw*.

GetData

```
procedure GetData(var Rec); virtual;
```

Diese pseudo-abstrakte Methode kopiert *Value* in *Rec*. Sie müssen diese Methode bei von *TCluster* abgeleiteten Objekten überschreiben, um zu gewährleisten, daß die Methoden *DataSize* und *SetData* korrekt arbeiten. Siehe auch: *TCluster.DataSize*, *TCluster.SetData*.

GetHelpCtx

```
function GetHelpCtx: Word; virtual;
```

Gibt den Wert aus *HelpCtx* plus *Sel* zurück. Dies erlaubt Ihnen, für jede Option des Schaltfeldes eine eigene Hilfenummer vorzusehen. Allerdings müssen Sie dann einen Hilfenummernbereich in der Größe *HelpCtx* plus Zahl der Optionen minus eins für dieses Schaltfeld reservieren.

GetPalette

```
function GetPalette: PPalette; virtual;
```

Diese pseudo-abstrakte Methode gibt einen Zeiger auf die Farbpalette *CCluster* zurück.

HandleEvent

```
procedure HandleEvent(var Event: TEvent; virtual;
```

Ruft zunächst *TView.HandleEvent* auf und arbeitet dann alle Ereignisse ab, die das Schaltfeld selbst betreffen. Die einzelnen Schalter dieses Feldes können sowohl mittels der Maus als auch über die Cursortasten angewählt und mit [Leertaste] markiert werden. Anschließend wird das Schaltfeld mit den schon gewählten Optionen neu dargestellt. Siehe auch: *TView.HandleEvent*.

Mark

```
function Mark(Item: Integer): Boolean; virtual;
```

Diese pseudo-abstrakte Methode gibt False zurück. Sie sollten sie überschreiben, damit diese Methode immer dann True zurückgibt, wenn der Schalter *Item* des Schaltfelds markiert werden muß.

MovedTo

```
procedure MovedTo(Item: Integer); virtual;
```

Bewegt die Selektionsmarkierung auf den Schalter *Item* des Schaltfeldes. *MovedTo* ist eine abstrakte Methode, muß also überschrieben werden.

Press

```
procedure Press(Item: Integer); virtual;
```

Wird immer dann aufgerufen, wenn der Schalter *Item* des Schaltfelds mittels der Maus oder über die Tastatur »gedrückt« wurde. *Press* ist eine abstrakte Methode, muß also überschrieben werden.

SetData

```
procedure SetData(var Rec); virtual;
```

Diese pseudo-abstrakte Methode liest *Value* aus *Rec* und ruft darauf *TView.DrawView* auf, die das Schaltfeld auf dem Bildschirm darstellt. Sie müssen diese Methode bei von *TCluster* abgeleiteten Objekten überschreiben, um zu gewährleisten, daß die Methoden *DataSize* und *SetData* korrekt arbeiten. Siehe auch: *TCluster.DataSize*, *TCluster.GetData*, *TView.DrawView*.

SetState

```
procedure SetState(AState: Word;
                   Enable: Boolean); virtual;
```

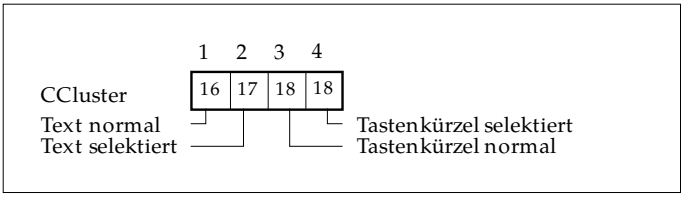
Ruft *TView.SetState*(*AState*, *Enable*) und, sofern *AState* den Wert *sfSelected* aufweist, anschließend *TView.DrawView* zur Darstellung des Schaltfeldes auf. Siehe auch: *TView.SetState*, *TView.DrawView*.

Store

```
procedure Store(var S: TStream);
```

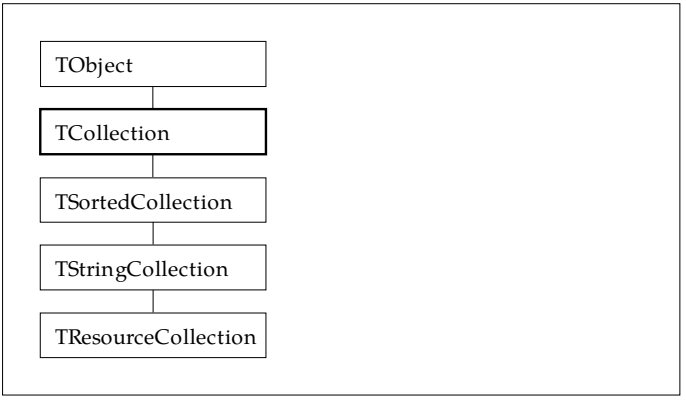
Speichert ein *TCluster*-Objekt über *TView.Store*(*S*) im angegebenen Stream. Diese Methode wird zusammen mit *TButton.Load* genutzt, um Objekte des Typs *TButton* in einem Stream zu speichern bzw. wieder zu laden. Siehe auch: *TView.Store*, *TButton.Load*, *TStream.Write*.

Farbpalette Dieses Objekt greift über die Palette *CCluster* auf die Einträge 16 bis 18 der Palette *CDialog* zu.



TCollection OBJECTS

TCollection bietet eine allgemeine Grundlage zur Implementierung eines »Behälters« für Daten und Objekte. Das Konzept, das *TCollection* zugrundeliegt, ist wesentlich flexibler als das von Arrays, Sets oder Listen. Eine Kollektion paßt sich zur Laufzeit dynamisch an die Größe der ihr übergebenen Daten an und bildet die Grundlage für spezialisiertere



Objekttypen, wie *TSortedCollection*, *TStringCollection* und *TResourceCollection*. Das Objekt verfügt über Methoden, die erlauben, Einträge hinzuzufügen oder Einträge zu löschen. Darüber hinaus stehen auch Wiederholmethoden bereit, die eine vom Anwender zu definierende Funktion auf jeden Eintrag in der Kollektion anwenden.

Felder

Items

```
Items: PItemList;
```

Ein Zeiger auf ein Array, das seinerseits wieder Zeiger auf die Dateneinträge speichert.

Count

```
Count: Integer;
```

Speichert die aktuelle Anzahl der Einträge in der Kollektion. Siehe auch: *MaxCollectionSize*.

Limit

```
Limit: Integer;
```

Die zunächst gültige Maximalanzahl von Einträgen in der Kollektion. Siehe auch: *Delta*, *TCollection.Init*.

Delta

```
Delta: Integer;
```

Die Anzahl von Einträgen, die eine Kollektion über die in *Limit* gesetzte Grenze hinaus aufnehmen darf. Wird dieses Feld zum Beispiel auf den Wert 0 gesetzt, darf die Kollektion nicht mehr Einträge aufnehmen, als über *Limit* angegeben wurde.

Hinweis: Es ist ziemlich zeitaufwendig, die Größe einer Kollektion zu ändern. Achten Sie daher darauf, den Wert für *Limit* so zu wählen, daß er in der Regel alle Einträge einschließt, die Sie in einer Kollektion speichern wollen. Setzen Sie dann *Delta* auf einen Wert, der alle Eventualitäten abdeckt. Siehe auch: *Limit*, *TCollection.Init*.

Methoden

Init

```
constructor Init(ALimit, ADelta: Integer;
```

Initialisiert eine Kollektion, deren Feld *Limit* auf *Alimit* und deren Feld *Delta* auf *ADelta* gesetzt ist. Zunächst ist die Zahl der Einträge auf *ALimit* begrenzt; ingesamt kann die Kollektion aber *ALimit* + *ADelta* Einträge aufnehmen. Siehe auch: *TCollection.Limit*, *TCollection.Delta*.

Load

```
constructor Load(var S: TStream);
```

Inialisiert und lädt eine Kollektion aus dem angegebenen *Stream*. *TCollection.Load* ruft für jeden Eintrag der Kollektion die Methode *GetItem* auf. Siehe auch: *TCollection.GetItem*.

Done

```
destructor Done; virtual;
```

Löscht über *TCollection.FreeAll* alle Einträge der Kollektion und gibt den von ihnen belegten Speicher frei; das Feld *Limit* der Kollektion wird auf 0 gesetzt. Siehe auch: *TCollection.FreeAll*, *TCollection.Init*.

At

```
function At(Index: Integer): Pointer;
```

Gibt einen Zeiger auf den durch *Index* spezifizierten Eintrag zurück. Über diese Methode können Sie auf eine Kollektion so zugreifen, als sei sie ein indiziertes Array. Ist *Index* kleiner 0 oder größer als bzw. gleich *Count*, wird die Methode *Error* mit dem Fehlerwert *coIndexErr* aufgerufen. In diesem Fall gibt *At* **nil** zurück. Siehe auch: *TCollection.IndexOf*.

AtDelete

```
procedure AtDelete(Index: Integer);
```

Löscht in der Kollektion einen Eintrag an der durch *Index* angegebenen Position. Diesem Eintrag eventuell noch folgende Einträge werden um eine Indexposition nach vorn versetzt; die Größe des von der Kollektion belegten Speichers wird jedoch nicht reduziert. Ist *Index* kleiner 0 oder größer als bzw. gleich *Count*, wird die Methode *Error* mit dem Fehlerwert *coIndexErr* aufgerufen. Siehe auch: *TCollection.AtInsert*, *TCollection.AtPut*.

AtInsert

```
procedure AtInsert(Index: Integer; Item: Pointer);
```

Fügt den Eintrag *Item* an der über *Index* angegebenen Position ein. Auf diese Position noch folgende Einträge werden nach hinten versetzt und neu indiziert. Ist *Index* kleiner 0 oder größer als bzw. gleich *Count*, wird die Methode *Error* mit dem Fehlerwert *coIndexErr* aufgerufen und der neue Eintrag nicht aufgenommen. War *Count* gleich *Limit* bevor *AtInsert* aufgerufen wurde, wird versucht, die Kollektion um *Delta* Einträge zu erweitern. Gelingt dies nicht, wird die Methode *Error* mit dem Fehlerwert *coOverflow* aufgerufen und der neue Eintrag

nicht aufgenommen. Siehe auch: *TCollection.At*, *TCollection.AtPut*.

AtPut

```
procedure AtPut(Index: Integer; Item: Pointer);
```

Ersetzt den Eintrag an der über *Index* angegebenen Position durch den Eintrag *Item*. Ist *Index* kleiner 0 oder größer als bzw. gleich *Count*, wird die Methode *Error* mit dem Fehlerwert *coIndexErr* aufgerufen. Siehe auch: *TCollection.At*, *TCollection.AtInsert*.

Delete

```
procedure Delete(Item: Pointer);
```

Löscht den Eintrag, auf den *Item* zeigt, gibt aber den Zeiger auf diesen Eintrag nicht frei. Daher wird auch die Indizierung der anderen Einträge nicht geändert. Der Aufruf dieser Methode entspricht der Anweisung *AtDelete*(*IndexOf*(*Item*)). Siehe auch: *TCollection.AtDelete*, *TCollection.DeleteAll*.

DeleteAll

```
procedure DeleteAll;
```

Löscht alle Einträge in der Kollektion, gibt jedoch die Zeiger auf diese Einträge nicht frei. Siehe auch: *TCollection.AtDelete*, *TCollection.DeleteAll*.

Error

```
procedure Error(Code, Info: Integer); virtual;
```

Wird immer dann aufgerufen, wenn in der Kollektion ein Fehler auftrat. Sofern nicht überschrieben, löst diese Methode einen Laufzeitfehler mit der Nummer 212 aus.

FirstThat

```
function FirstThat(Test: Pointer): Pointer;
```

Test zeigt auf eine Funktion, die der Benutzer bereitstellen muß und die einen Wert des Typs *Boolean* zurückgibt. Diese Funktion wendet *TCollection.FirstThat* solange auf jeden Eintrag in der Kollektion an, bis sie *True* zurückgibt, wobei *FirstThat* mit dem ersten Eintrag beginnt. Die Methode gibt einen Zeiger auf diesen Eintrag zurück oder **nil**, wenn kein passender Eintrag gefunden wurde. Test muß auf eine *lokale*, als **far** deklarierte Funktion zeigen, die als Parameter einen Zeiger übernimmt und einen boolschen Wert zurückgibt:

```
function Matches(Item: Pointer): Boolean; far;
```

Lokal heißt, daß diese Funktion innerhalb der Prozedur oder Funktion definiert sein muß, innerhalb der auch die Methode *FirstThat* aufgerufen wird.

Nehmen wir an, *List* sei eine Instanz von *TCollection*. Die Anweisung

```
F := List.FirstThat(@Matches);
```

entspricht dann der Anweisungsfolge

```
I := 0;
while (I < List.Count) and not Matches(List.At(I))
  do Inc(I);
if I < List.Count then P := List.At(I) else P := nil;
```

Siehe auch: *TCollection.LastThat*, *TCollection.ForEach*.

ForEach

```
procedure ForEach(Action: Pointer);
```

Diese Wiederhol-Methode wendet die Routine, auf die *Action* zeigt, auf jeden Eintrag der Kollektion an. *Action* muß auf eine *lokale*, als **far** deklarierte Prozedur zeigen, die als Parameter einen Zeiger übernimmt:

```
procedure PrintItem(Item: Pointer); far;
```

Lokal heißt, daß diese Prozedur innerhalb der Prozedur oder Funktion definiert sein muß, innerhalb der auch die Methode *ForEach* aufgerufen wird.

Nehmen wir an, *List* sei eine Instanz von *TCollection*. Die Anweisung

```
List.ForEach(@PrintItem);
```

entspricht dann der Anweisungsfolge

```
for I := 0 to List.Count do PrintItem(List.At(I));
```

Siehe auch: *TCollection.FirstThat*, *TCollection.LastThat*.

Free

```
procedure Free(Item: Pointer);
```

Löscht den Eintrag, auf den *Item* zeigt, und gibt den belegten Speicher frei. Entspricht den Anweisungen:

```
FreeItem(Item);
Delete(Item);
```

Siehe auch: *TCollection.FreeItem*, *TCollection.Delete*.

FreeAll

```
procedure FreeAll;
```

Löscht alle Einträge in der Kollektion und gibt den von ihnen belegten Speicher frei. Siehe auch: *TCollection.DeleteAll*.

FreeItem

```
procedure FreeItem(Item: Pointer); virtual;
```

Diese Methode muß auch den von *Item^* belegten Speicher freigeben. Sofern nicht überschrieben, geht *FreeItem* davon aus, daß *Item* ein Zeiger auf ein von *TObject* abgeleitetes Objekt ist, und ruft dessen Destruktor auf:

```
if Item <> nil then Dispose(PObject(Item), Done);
```

FreeItem wird von den Methoden *Free* und *FreeAll* benutzt und sollte nie direkt aufgerufen werden. Siehe auch: *TCollection.Free*, *TCollection.FreeAll*.

GetItem

```
function GetItem(var S: TStream): Pointer; virtual;
```

Wird von *TCollection.Load* für jeden in dem Stream gespeicherten Eintrag aufgerufen. Diese Methode kann überschrieben, sollte aber nie direkt aufgerufen werden. Sofern nicht überschrieben, ruft *GetItem* die Methode *S.Get* auf. Siehe auch: *TStream.Get, TCollection.Load*.

IndexOf

```
function IndexOf(Item: Pointer): Integer; virtual;
```

Gibt einen Index auf den Eintrag *Item* zurück oder -1, wenn *Item* in der Kollektion nicht gefunden wurde. Siehe auch: *TCollection.At*.

Insert

```
procedure Insert(Item: Pointer); virtual;
```

Fügt *Item* in die Kollektion ein und berichtigt – sofern notwendig – die Indizierung der vorhandenen Einträge. Sofern nicht überschrieben, ruft diese Methode *AtInsert*(*Item, Count*) auf, wodurch der neue Eintrag an das Ende der Kollektion angehängt wird. Siehe auch: *TCollection.AtInsert*.

LastThat

```
function LastThat(Test: Pointer): Pointer;
```

Test zeigt auf eine Funktion, die der Benutzer bereitstellen muß, und die einen Wert des Typs *Boolean* zurückgibt. Diese Funktion wendet *TCollection.LastThat*, beginnend mit dem letzten Eintrag, auf jeden Eintrag der Kollektion an. Die Methode gibt einen Zeiger auf den Eintrag zurück, für den die Funktion *True* ermittelt, oder **nil**, wenn kein solcher Eintrag gefunden wurde. *Test* muß auf eine *lokale*, als **far** deklarierte Funktion zeigen, die als Parameter einen Zeiger übernimmt und einen boolschen Wert zurückgibt:

```
function Matches(Item: Pointer): Boolean; far;
```

Lokal heißt, daß diese Funktion innerhalb der Prozedur oder Funktion definiert sein muß, innerhalb der auch die Methode *FirstThat* aufgerufen wird. Nehmen wir an, *List* sei eine Instanz von *TCollection*. Die Anweisung

```
P := List.LastThat(@Matches);
```

entspricht dann der Anweisungsfolge

```
I := List.Count - 1;
while (I >= 0) and not Matches(List.At(I)) do Dec(I);
if I >= 0 then P := List.At(I) else P := nil;
```

Siehe auch: *TCollectionFirstThat, TCollection.ForEach*.

Pack

```
procedure Pack;
```

Löscht alle **nil**-Zeiger in der Kollektion. Siehe auch: *TCollection.Delete, TCollection.DeleteAll*.

PutItem

```
procedure PutItem(var S: TStream;
                  Item: Pointer); virtual;
```

Wird von *TCollection.Store* für jeden Eintrag der Kollektion aufgerufen. Diese Methode kann überschrieben, sollte aber nie direkt aufgerufen werden. Sofern nicht überschrieben, geht *PutItem* davon aus, daß die Einträge in der Kollektion von *TObject* abgeleitete Objekt sind und ruft die Methode *S.Put*(*Item*) auf. Siehe auch: *TStream.Put*, *TCollection.GetItem*, *TCollection.Store*, *TCollection.Load*.

SetLimit

```
procedure SetLimit(ALimit: Integer); virtual;
```

Ändert die maximale Größe der Kollektion. Ist *ALimit* kleiner Count, erhält *ALimit* den Wert *Count*. Ist *ALimit* größer *MaxCollectionSize*, erhält *ALimit* den Wert *MaxCollectionSize*. Unterscheidet sich dann *ALimit* von *Limit*, wird Speicher für eine Kollektion mit *ALimit* Einträgen reserviert, der Inhalt der alten Kollektion in die neue kopiert und dann der Speicher freigegeben, den die alte Kollektion belegt. Siehe auch: *Limit*, *Count*, *MaxCollectionSize*.

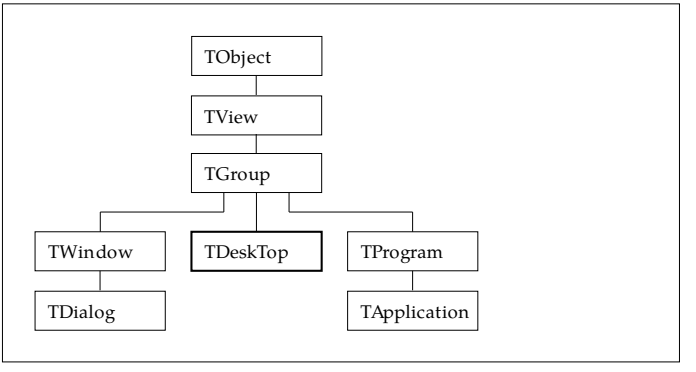
Store

```
procedure Store(var S: TStream);
```

Speichert die Kollektion einschließlich ihrer Einträge im angegebenen Stream. Dabei ruft *TCollection.Store* für jeden Eintrag die Methode *PutItem* auf. Siehe auch: *TCollection.Load*, *TCollection.PutItem*.

TDeskTop

APP



TDeskTop ist eine recht einfache Gruppe, zu der nur ein *TBackGround*-Objekt gehört, das den Hintergrund bildet, vor dem die anderen Ausgaben des Programms erscheinen. Ein

TDeskTop-Objekt verwaltet den gesamten Bildschirmbereich zwischen der Menüleiste und der Statuszeile.

Methoden

Init `constructor Init(var Bounds: TRect);`

Initialisiert eine *TDeskTop*-Gruppe mit der in *Bounds* angegebenen Größe und setzt die Bits *gfGrowHiX* und *gfGrowHiY* in *GrowMode*. Darauf ruft dieser Konstruktor *NewBackground* auf und fügt ein *TBackground*-Objekt in die Gruppe ein. Siehe auch: *TDeskTop.NewBackGround*, *TGroup.Init*, *TGroup.Insert*.

Cascade `procedure Cascade(var Bounds: TRect);`

Stellt die Fenster, die zur *TDeskTop*-Gruppe gehören und über die Eigenschaft *ofTileable* verfügen, fächerartig angeordnet auf dem Bildschirm dar. Siehe auch: *TDeskTop.Tile*.

NewBackground `function NewBackground: PView; virtual;`

Gibt einen Zeiger auf das *TBackGround*-Objekt zurück, das mit *TDeskTop* verbunden ist. Genutzt wird diese Methode von *TDeskTop.Init*; abgeleitete Objekte können durch Überschreiben dieser Methode ein anderes Objekt einbinden. Siehe auch: *TDeskTop.Init*.

HandleEvent `procedure HandleEvent(var Event: TEvent); virtual;`

Diese Methode ruft *TGroup.HandleEvent* auf und ist zuständig für die Bearbeitung der Befehle *cmNext* (der zum Beispiel in der integrierten Entwicklungsumgebung über [F6] ausgelöst wird) und *cmPrevious*. Dabei bearbeitet diese Methode, beginnend mit dem Objekt, das die Eigenschaft *sfSelected* besitzt, nacheinander die zu *TDeskTop* gehörenden Objekte. Siehe auch: *TGroup.HandleEvent*, *cmNext*, *TGroup.SelectNext*.

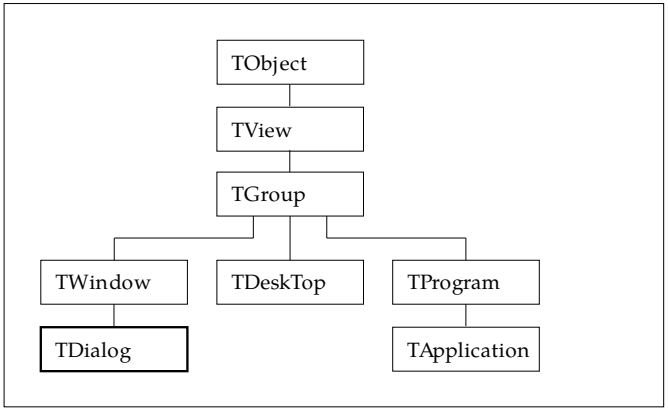
Tile `procedure Tile(var R: TRect);`

Stellt die Fenster, die zur TDeskTop-Gruppe gehören und über die Eigenschaft *ofTileable* verfügen, neben- und untereinander angeordnet auf dem Bildschirm dar. Siehe auch: *TDeskTop.Cascade*.

TileError `procedure TileError; virtual;`

Wird immer dann aufgerufen, wenn bei der Ausführung von *Tile* oder *Cascade* ein Fehler auftrat. Allerdings beschränkt sich diese Methode darauf, nichts zu tun. Sie müssen Sie überschreiben, wenn Sie zum Beispiel den Benutzer auf einen solchen Fehler aufmerksam machen wollen. Siehe auch: *TDeskTop.Tile*, *TDeskTop.Cascade*.

TDialog



TDialog verfügt über einige Eigenschaften, die dieses Objekt von seinem unmittelbaren Vorfahren *TWindow* unterscheiden. Wie dieses kann auch *TDialog* über den Bildschirm bewegt werden, kann jedoch nicht in seiner Größe geändert und damit auch nicht gezoomt werden. Die *Valid*-Methode gibt auf jeden Fall True zurück, wenn der übergebene Befehl *cmCancel* ist, und ruft erst dann *TGroup.Valid* auf, wenn dies nicht der Fall ist.

Methoden

Init

```
constructor Init(var Bounds: TRect; ATitle: TTitleStr);
```

Initialisiert über *TWindow.Init*(*Bounds, ATitle, wnNoNumber*) ein Dialogfenster in der in *Bounds* angegebenen Größe. *GrowMode* wird auf 0 und in Flags die Bits *wfMove* und *wfClose* gesetzt, was bedeutet, daß dieses Fenster versetzt und geschlossen, aber nicht vergrößert oder verkleinert werden kann. Beachten Sie, daß *TDialog* keinen eigenen Destruktor definiert, sondern die von *TView*, *TGroup* und *TWindow* geerbten Methoden *Close* und *Done* benutzt.

Siehe auch: *TWindowp.Init*, *TWindow.Close*.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Ruft *TWindow.HandleEvent* auf, behandelt selbst aber die Tastaturereignisse »[Enter] gedrückt« und »[Esc] gedrückt« auf spezielle Weise. Das erste Ereignis generiert den Befehl *cmDefault*, das zweite den Befehl *cmCancel*. Weiterhin reagiert diese Methode auf die Befehle *cmOK*, *cmYes* und *cmNo*, die zusammen mit den schon genannten dazu führen, daß die

TDeskTop-Gruppe die Eigenschaft der Modalität abgibt. Siehe auch: *TWindow.HandleEvent*.

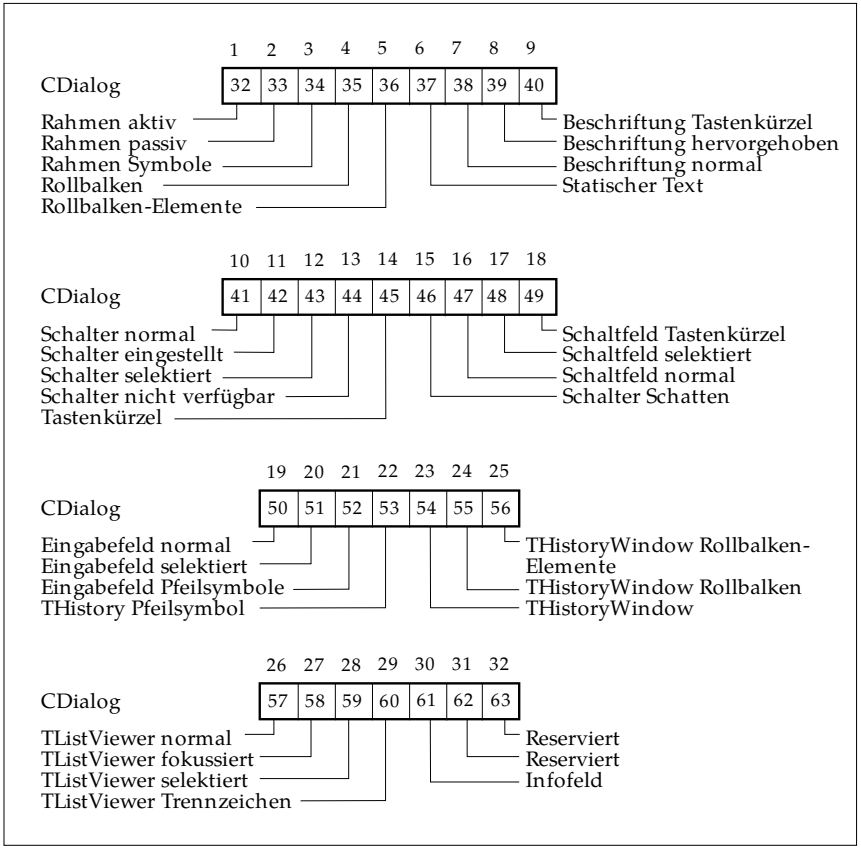
GetPalette `function GetPalette: PPalette; virtual;`

Diese Methode gibt einen Zeiger auf die Farbpalette *CDialog* zurück.

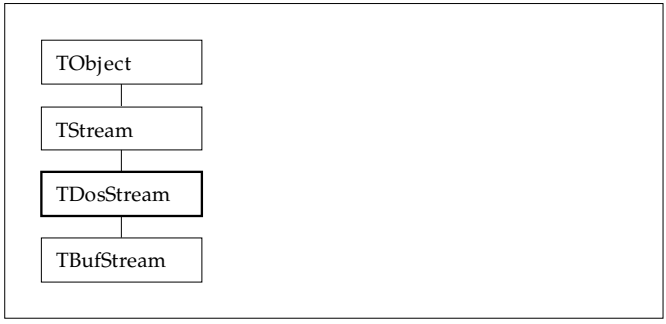
Valid `function Valid(Command: Word): Boolean; virtual;`

Gibt True zurück, wenn entweder der in *Command* übergebene Befehl *cmCancel* ist oder alle gleichnamigen Methoden der zur Gruppe gehörenden Objekte True ermitteln. Siehe auch: *TGroup.Valid*.

Farbpalette Dieses Objekt greift über die Palette *CDialog* auf die Einträge 32 bis 63 der aktuellen Farbpalette des Programms zu.



TDosStream OBJECTS



TDosStream ist ein spezialisierter Nachkomme von *TStream* und implementiert die Ein- und Ausgabe polymorpher Objekte in bzw. aus einer DOS-Datei. Diese Ein- und Ausgaben werden nicht über einen Puffer abgewickelt, weshalb die meisten Programme sich auf einen Nachkommen dieses Objekts, nämlich *TBufStream* stützen, das einen solchen Puffer verwendet. Die Schlüsselmethode von *TDosStream* ist der Konstruktor *Init*, der erlaubt, je nach dem Wert seines Parameters *Mode* eine DOS-Datei zu erstellen oder eine schon vorhandene zu öffnen. Das einzige Feld, das *TDosStream* definiert, ist der Datei-Handle, über den der Zugriff auf die Datei abgewickelt wird. Dieses Objekt überschreibt alle abstrakten Methoden von *TStream*, ausgenommen dessen Methode *Flush*.

Felder

Handle

```
Handle: Word;
```

Dieses Feld speichert den Datei-Handle, über den der Zugriff auf eine geöffnete DOS-Datei abgewickelt wird.

Methoden

Init

```
constructor Init(FileName: FNameStr; Mode: Word);
```

In Abhängigkeit von dem Parameter *Mode* wird eine Datei unter dem Namen *FileName* neu angelegt oder eine solche, schon existierende Datei geöffnet. Ist diese Operation erfolgreich, enthält *Handle* den Datei-Handle und das Feld *ErrorInfo* den Wert *stOK*, im Fehlerfall dagegen den Wert *stInitError*.

Das Argument *Mode* kann einen der folgenden Werte annehmen: *stCreate*, *stOpenRead*, *stOpenWrite* oder *stOpen*. Diese Konstanten sind in Kapitel 15, »Außerhalb der Objekte«, unter dem Stichwort »*stXXXX Stream-Konstanten*« näher erläutert.

Done `destructor Done; virtual;`

Schließt die Datei und gibt den von dem Stream belegten Speicher frei. Siehe auch: *TDosStream.Init*.

GetPos `function GetPos: LongInt; virtual;`

Gibt die aktuelle Position im Stream zurück. Siehe auch: *TStream.Get*, *TStream.Put*.

GetSize `function GetSize: LongInt; virtual;`

Gibt die Größe des Stream in Bytes zurück. Siehe auch: *TStream.CopyFrom*.

Read `procedure Read(var Buf; Count: Word); virtual;`

Liest, beginnend an der aktuellen Position im Stream, *Count* Bytes aus dem Stream und speichert sie in *Buf*. Siehe auch: *stReadError*, *TDosStream.GetPos*, *TDosStream.Seek*.

Seek `procedure Seek(Pos: LongInt); virtual;`

Diese Methode setzt die aktuelle Position auf die Stelle *Pos* Bytes, gezählt ab Beginn des Stream. Der Stream beginnt mit der Position 0. Siehe auch: *TDosStream.GetPos*, *TStream.Put*, *TStream.Get*, *TStream.CopyFrom*.

Truncate `procedure Truncate; virtual;`

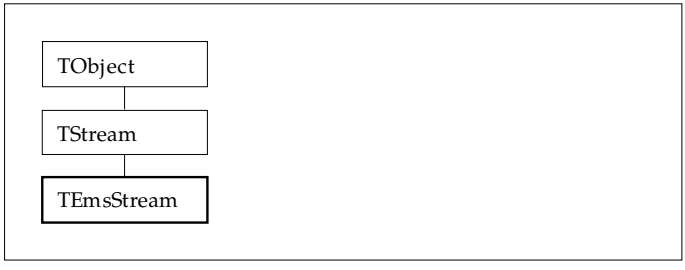
Löscht ab der aktuellen Position bis zum Ende des Stream alle Daten. Die Position selbst verändert sich nicht, zeigt also nach der Operation auf das Ende des Stream. Siehe auch: *TDosStream.GetPos*, *TDosStream.Seek*.

Write `procedure Write(var Buf; Count: Word); virtual;`

Schreibt, beginnend an der aktuellen Position im Stream, *Count* Bytes aus *Buf* in den Stream. Siehe auch: *TDosStream.Read*, *TDosStream.GetPos*, *TDosStream.Seek*.

TEmsStream

OBJECTS



TEmsStream ist ein spezialisierter Nachkomme von *TStream* und nutzt den EMS-Speicher zur Ablage polymorpher Objekte. Die zusätzlichen Felder, die *TEmsStream* definiert, speichern den EMS-Handle, führen Buch über die belegten Speicherseiten, die Größe des Stream und die aktuelle Position. Dieses Objekt überschreibt alle abstrakten Methoden von *TStream*, ausgenommen natürlich dessen Methode *Flush*, und definiert einen speziellen Konstruktor und Destruktor.

Felder

Handle

```
Handle: Word;
```

Speichert den EMS-Handle, über den der Zugriff auf den Speicher abgewickelt wird.

PageCount

```
PageCount: Word;
```

Zahl der von dem Stream im EMS-Speicher belegten Seiten.

Size

```
Size: LongInt;
```

Enthält die Größe des Stream in Bytes.

Position

```
Position: LongInt;
```

Speichert die aktuelle Position im Stream; die Startposition ist 0.

Methoden

Init

```
constructor Init(MinSize, MaxSize: LongInt);
```

EMS-Treiber vor der Version 4.0 unterstützen nicht die Änderung der Größe eines einmal im EMS-Memory reservierten Speicherblocks. Daher kann ein EMS-Stream nicht die bei der Initialisierung festgelegte Größe überschreiten, wenn ein solcher Treiber ermittelt wird. Daher übernimmt *Init*, um sowohl EMS-Treiber-Versionen älter als auch neuer als 4.0 zu unterstützen, die beiden Parameter *MinSize* und *MaxSize*, um die Anfangsgröße des zu reservierenden EMS-Speicherblocks zu bestimmen. *Init* reserviert zumindest einen Speicherblock der Größe *MinSize*.

Ist die Versionsnummer des EMS-Treibers größer oder gleich 4.0, reserviert *Init MinSize* Bytes Speicher und vergrößert anschließend diesen Block, wenn Aufrufe von *TEmsStream.Write* das notwendig machen sollten. In diesem Fall wird der Parameter *MaxSize* vollständig ignoriert.

Ist die Versionsnummer des EMS-Treibers kleiner als 4.0, reserviert *Init*, wenn verfügbar, bis zu *MaxSize* Bytes Speicher. Wenn Aufrufe von *TEmsStream.Write* eine Vergrößerung des Speicherblockes notwendig machen sollten, wird ein Fehler ausgelöst.

Nach der Speicherreservierung wird *TStream.Init* aufgerufen und die Felder *Handle*, *Size* und *PageCount* gesetzt. Enthält das Feld *ErrorInfo* den Wert *stInitError*, ist die Initialisierung fehlgeschlagen. Siehe auch: *TEmsStream.Done*.

Done `destructor Done; virtual;`

Gibt den belegten EMS-Speicher und dann auch den Speicher frei, den der Stream selbst belegt. Siehe auch: *TEmsStream.Init*.

GetPos `function GetPos: LongInt; virtual;`

Gibt die aktuelle Position im Stream zurück. Siehe auch: *TStream.Get*, *TStream.Put*.

GetSize `function GetSize: LongInt; virtual;`

Gibt die Größe des Stream in Bytes zurück. Siehe auch: *TStream.CopyFrom*.

Read `procedure Read(var Buf; Count: Word); virtual;`

Liest, beginnend an der aktuellen Position im Stream, *Count* Bytes aus dem Stream und speichert sie in *Buf*. Siehe auch: *stReadError*, *TEmsStream.GetPos*, *TEmsStream.Seek*.

Seek `procedure Seek(Pos: LongInt); virtual;`

Diese Methode setzt die aktuelle Position auf die Stelle *Pos* Bytes ab Beginn des Stream. Der Stream beginnt mit der Position 0. Siehe auch: *TEmsStream.GetPos*, *TStream.Put*, *TStream.Get*, *TStream.CopyFrom*.

Truncate `procedure Truncate; virtual;`

Löscht ab der aktuellen Position bis zum Ende des Stream alle Daten. Die Position selbst verändert sich nicht, zeigt also nach der Operation auf das Ende des Stream. Siehe auch: *TEmsStream.GetPos*, *TEmsStream.Seek*.

Write `procedure Write(var Buf; Count: Word); virtual;`

Schreibt, beginnend an der aktuellen Position im Stream, *Count* Bytes aus *Buf* in den Stream. Siehe auch: *TEmsStream.Read*, *TEmsStream.GetPos*, *TEmsStream.Seek*.

TFrame

VIEWS

Dieses Objekt versieht Bildschirmfenster mit einem Rahmen. Sie werden das Objekt *TFrame* wahrscheinlich nicht direkt benutzen, da *TWindow*-Objekte automatisch mit einem Rahmen versehen werden.

Methoden

Init

```
constructor Init(var Bounds: TRect);
```

Initialisiert über *TView.Init*(*Bounds*) einen Rahmen in der in *Bounds* angegebenen Größe. In *GrowMode* werden die Bits *gfGrowHiX* und *gfGrowHiY* gesetzt, und *EventMask* erhält den Wert (*EventMask* **or** *evBroadCast*), damit *TFrame*-Objekte auf Ereignisse des Typs *evBroadCast* reagieren. Siehe auch: *TView.Init*.

Draw

```
procedure Draw; virtual;
```

Gibt den Rahmen auf dem Bildschirm aus. Die Art und die verwendete Farbe werden bestimmt durch den in *State* festgehaltenen Status des mit diesem Objekt verbundenen Fensterobjekts. Kann dieses gezoomt und geschlossen werden, werden auch die entsprechenden Icons dargestellt. Auch die Ausgabe des Titels – sofern das verbundene Fenster einen solchen definiert – fällt in den Aufgabenbereich dieser Methode. Siehe auch: *TView.Draw*, *sfXXXX*-Konstanten, *wfXXXX*-Konstanten.

GetPalette

```
function GetPalette: PPalette; virtual;
```

Diese pseudo-abstrakte Methode gibt einen Zeiger auf die Farbpalette *CFrame* zurück.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Ruft *TView.HandleEvent* auf, ist aber selbst zuständig für Mausereignisse, die Form und Position des zugeordneten Fensterobjekts betreffen. Siehe auch: *TView.HandleEvent*.

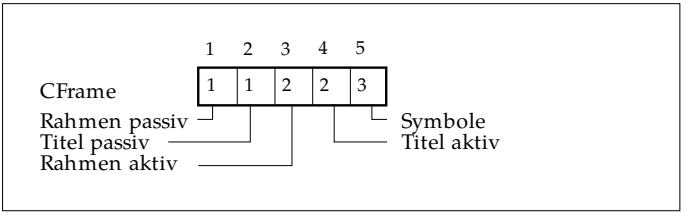
SetState

```
procedure SetState(AState: Word;
                   Enable: Boolean); virtual;
```

Ruft *TView.SetState*(*AState, Enabled*) auf. Schließt der neue Status die Eigenschaften *sfActive* oder *sfDragging* ein, wird *DrawView* aufgerufen, das Objekt also neu dargestellt. Siehe auch: *TView.SetState*.

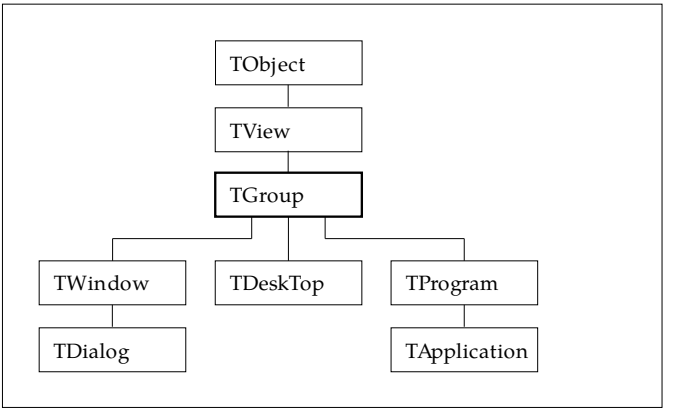
Farbpalette

Dieses Objekt greift über die Palette *CFrame* auf die Einträge 1 bis 3 der Palette *CWindow* zu.



TGroup

VIEWS



Das Objekt *TGroup* und seine Nachkommen, nennen wir der Kürze wegen *Gruppen*, und solche Gruppen sind die zentralen Elemente in Turbo Vision. Ihrer Abstammung entsprechend ist auch eine Gruppe nichts anderes als ein *View*-Objekt, unterscheidet sich von diesem aber durch eine herausragende Eigenschaft: Ein *TGroup*-Objekt verwaltet eine dynamische, verkettete Liste anderer *View*-Objekte. Objekte, die zu einer Gruppe gehören, nennen wir *Subviews* dieser Gruppe, das auch dann, wenn sie ihrerseits selbst Gruppen von Objekten sind. *TGroup* überschreibt viele, von *TView* geerbte Methoden, um zu gewährleisten, daß bestimmte Operationen auf eine Gruppe ebenso angewendet werden können, als handele es sich dabei um ein einzelnes Objekt. So erlauben es die Methoden *TGroup.Load* und *TGroup.Store* beispielsweise, eine ganze Gruppe von Objekten aus einem Stream zu laden oder sie in einem Stream zu speichern.

Auch einer Gruppe wird ein rechteckiger Bildschirmausschnitt zugewiesen, den sie verantwortlich zu verwalten hat. Jedoch delegiert sie diese Aufgabe weiter an die *View*-Objekte, die zu ihr gehören. Eines dieser Objekte muß in der Lage

sein, den Bildschirmbreich der Gruppe zu überdecken. In der Konsequenz heißt das: Auf dem Bildschirm ist die Gruppe nur über die Darstellung der *Subviews*, die zu ihr gehören, sichtbar (darauf gehen wir näher ein in Kapitel 4, »View-Objekte«). Auch für die Bearbeitung von Ereignissen sind, wie in Kapitel 5, »Ereignisgesteuerte Programme«, dargestellt, diese *Subviews* zuständig. Während des Programmlaufs werden neue *Subviews* kreiert, in eine Gruppe eingefügt, auf dem Bildschirm dargestellt, wieder vom Bildschirm und aus der Gruppe entfernt, entsprechend den Befehlen, die der Benutzer mittels der Maus oder über die Tastatur aktiviert.

Die drei in Turbo Vision definierten Nachkommen von *TGroup*, nämlich *TWindow*, *TDeskTop* und *TApplication* illustrieren das Konzept, das hinter einer Gruppe steht: Zu einem *TApplication*-Objekt (dessen Abstammung von *TGroup* über *TProgram* vermittelt wird) gehört üblicherweise ein *TDeskTop*-, ein *TStatusLine*- und ein *TMenuView*-Objekt. Nun ist aber *TDeskTop* selbst wieder ein Nachkomme von *TGroup* und kann sich zum Beispiel ein oder mehrere *TWindow*-Objekte »aneignen«, von denen jedes wiederum über ein TFrame-Objekt und ein *TScrollBar*-Objekt verfügen kann.

Normalerweise werden Sie keine Instanz von *TGroup* selbst deklarieren, sondern eher seine Nachkommen *TApplication*, *TDeskTop* und *TWindow* verwenden.

Felder

Last

```
Last: PView;
```

Zeigt auf die letzte Subview der Gruppe, das ist die Subview, die am Ende der durch die Z-Ordnung festgelegten Reihenfolge steht. Das Feld *Next* der letzten Subview zeigt auf die erste Subview; deren Feld *Next* auf die nächste Subview usw. Dadurch entsteht eine zirkulär verkettete Liste von Subviews.

Current

```
Current: PView;
```

Zeigt auf die aktuell selektierte Subview. Siehe auch: *sfSelected*.

Buffer

```
Buffer: PVideoBuf;
```

Sofern nicht **nil**, zeigt *Buffer* auf einen Puffer, der dazu dient, einen »Bildschirmabzug« des Bereichs, den die Gruppe verwaltet, aufzunehmen. Siehe auch: *TGroup.Lock*, *TGroup.Unlock*, *TGroup.DrawView*.

Phase

```
Phase: (phFocused, phPreProcess, phPostProcess);
```

Speichert die aktuelle Phase der Bearbeitung eines fokussierten Ereignisses. Jede Subview kann, sofern *ofPreProcess* oder *ofPostProcess* gesetzt ist, über das Feld *Owner^.Phase* erfahren, ob der Aufruf ihrer *HandleEvent*-Methode in der Phase *phPre-*

Process, *phFocused* oder *phPostProcess* erfolgt ist. Siehe auch: *ofPreProcess*, *ofPostProcess*, *TGroup.HandleEvent*.

Methoden

Init `constructor Init(var Bounds: TRect);`

Ruft *TView.Init*(*Bounds*) auf, um das Objekt mit der in *Bounds* angegebenen Größe zu initialisieren, und setzt die Begrenzungen, bei deren Überschreiten Subviews abgeschnitten werden. Im Feld *Options* werden die Bits *ofSelectable* und *ofBuffered* gesetzt; dem Feld *EventMask* wird der Wert $FFFF zugewiesen, wodurch das Objekt seine Zuständigkeit für alle Arten von Ereignissen erklärt. Siehe auch: *TGroup.Load*, *TView.Init*.

Load `constructor Load(var S: TStream);`

Lädt eine ganze Gruppe aus dem angegebenen Stream und ruft dazu zunächst *TView.Load* auf, benutzt dann *Stream.Get*, um die zur Gruppe gehörenden Subviews einzulesen. Danach werden alle Zeigerreferenzen, die über *GetPeerViewPtr* aus dem Stream gelesen wurden, aufgelöst. Wenn ein *TGroup*-Objekt Felder enthält, die auf Subviews zeigen, muß das Objekt innerhalb der *Load*-Methode *GetSubViewPtr* aufrufen, um diese Felder einlesen zu lassen. Siehe auch: *TGroup.Init, TGroup.Store, TGroup.GetSubViewPtr*.

Done `destructor Done; virtual;`

Gibt die in der Gruppe enthaltenen Objekte frei und entfernt sie vom Bildschirm. Ist ein Puffer angelegt, wird auch der von diesem Puffer belegte Speicher freigegeben. Siehe auch: *TView.Done*.

ChangeBounds `procedure ChangeBounds(var Bounds: TRect); virtual;`

Ändert die Begrenzungen der Gruppe auf die in *Bounds* übergebenen Werte und veranlaßt, daß alle von dieser Größenänderung betroffenen Subviews neu ausgegeben werden. Diese Methode überschreibt *TView.ChangeBounds*. Siehe auch: *TView.CalcBounds*, *DrawTView.ChangeBounds*.

DataSize `function DataSize: Word; virtual;`

Ruft für jede Subview deren gleichnamige Methode auf, addiert die zurückgegebenen Werte und ermittelt auf diese Weise die Größe des Datenbereichs der Gruppe. Siehe auch: *TView.DataSize*.

Delete `procedure Delete(P: PView );`

Sofern das Objekt, auf das *P* zeigt, sichtbar ist, wird es vom Bildschirm und danach auch aus der Gruppe entfernt; den

Feldern *Owner* und *Next* dieser Subview wird der Wert **nil** zugewiesen. Siehe auch: *TGroup.Insert*.

Draw

```
procedure Draw; virtual;
```

Sofern ein Puffer mit einem Abbild der Gruppe existiert, wird dessen Inhalt in den Bildschirmspeicher kopiert, ansonsten wird für jede Subview die Methode *TGroup.Redraw* aufgerufen. Siehe auch: *TGroup.Buffer*, *TGroup.Redraw*.

EndModal

```
procedure EndModal(Command: Word); virtual;
```

Ist die Gruppe das gerade modale *View*-Objekt, verliert es diese Eigenschaft. *Command* wird der Methode *ExecView* (die dieser Gruppe die Modalität zugeordnet hat) übergeben, die *Command* als Funktionsresultat zurückgibt. Ist die Gruppe *nicht* das modale *View*-Objekt, wird dagegen *TView.EndModal* aufgerufen. Siehe auch: *TGroup.ExecView*, *TView.EndModal*.

EventError

```
procedure EventError(var Event: TEvent); virtual;
```

Diese Methode wird immer dann aufgerufen, wenn die Methode *TGroup.Execute* auf ein Ereignis trifft, das sie nicht abarbeiten kann. Sofern nicht überschrieben, überprüft *EventError* zunächst, ob das Feld *Owner* auf **nil** gesetzt ist. Ist das nicht der Fall, wird das fragliche Ereignis an die *EventError*-Methode der übergeordneten Gruppe weitergeleitet, was möglicherweise bedeutet, daß sich schließlich die *EventError*-Methode des Objekts *TApplication* mit diesem Ereignis befassen muß. Sie müssen diese Methode also überschreiben, wenn *EventError* eine davon abweichende Aktion auslösen soll. Siehe auch: *TGroup.Execute*, *TGroup.ExecView*, *sfModal*.

ExecView

```
function ExecView(P: PView): Word;
```

Ist *P* **nil**, gibt *ExecView* den Befehl *cmCancel* zurück, andernfalls werden die Felder *Current*, *HelpContext* und *Command* gesichert, darauf P in die Gruppe eingefügt und dieses Objekt über einen Aufruf von *P^.Execute* zur modalen Subview erklärt. Nach der Rückkehr aus dieser Methode wird das Objekt aus der Gruppe entfernt und die oben genannten Felder wieder mit den gesicherten Werten belegt. Siehe auch: *TGroup.Execute*, *sfModal*.

Execute

```
function Execute: Word; virtual;
```

Diese Methode besteht im Prinzip aus zwei Programmschleifen, wobei die innere Schleife Ereignisse entgegennimmt und sie an *HandleEvent* weitergibt. Sie wird verlassen, wenn die Gruppe oder eine Subview *EndModal* aufruft. Vorher jedoch vergewissert sich diese Methode über *Valid*, daß der modale Zustand tatsächlich beendet werden kann.

Der folgende Programmausschnitt zeigt, wie *TGroup.Exe-*

cute implementiert ist. Beachten Sie, daß *EndState* ein als **private** deklariertes Feld ist, dem die Methode *EndModal* Werte zuweist:

```
function TGroup.Execute: Word;
var
  E: TEvent;
begin
  repeat
    EndState := 0;
    repeat
      GetEvent(E);
      HandleEvent(E);
      if E.What <> evNothing then EventError(E);
    until EndState <> 0
  until Valid(EndState);
  Execute := EndState;
end;
```

Siehe auch: *TGroup.GetEvent, TGroup.HandleEvent, TGroup.-EndModal, TGroup.Valid.*

First

```
function First: PView;
```

Gibt einen Zeiger auf die erste Subview der Gruppe zurück (also jene, die am Anfang der durch die Z-Ordnung festgelegten Reihenfolge steht) oder **nil**, wenn die Gruppe keine Subviews besitzt.

FirstThat

```
function FirstThat(Test: Pointer): PView;
```

Test zeigt auf eine Funktion, die einen boolschen Wert zurückgibt, und die Sie definieren müssen. In der Reihenfolge, die durch die Z-Ordnung festgelegt wird, wendet *FirstThat* diese Funktion auf jede Subview an, bis sie für eine Subview True zurückgibt. *Test* muß auf eine *lokale*, als **far** deklarierte Funktion zeigen, die als Parameter einen Zeiger übernimmt und einen boolschen Wert zurückgibt:

```
function MyTestFunc(P: PView): Boolean; far;
```

Die im folgenden Ausschnitt gezeigte Methode *SubViewAt* gibt einen Zeiger auf die erste Subview zurück, in deren Bereich der gesuchte Punkt enthalten ist:

```
function TMyGroup.SubViewAt(Where: TPoint): PView;

  function ContainsPoint(P: PView): Boolean; far;
  var
    Bounds: TRect;
  begin
    P^.GetBounds(Bounds);
```

```
    ContainsPoint := (P^.State and sfVisible <> 0) and
      Bounds.Contains(Where);
  end;

begin
  SubViewAt := FirstThat(@ContainsPoint);
end;
```

Siehe auch: *TGroup.ForEach*.

ForEach

```
procedure ForEach(Action: Pointer);
```

Action zeigt auf eine Funktion oder Prozedur, die *ForEach* auf jede Subview der Gruppe anwendet und zwar in der Reihenfolge, die durch die Z-Ordnung festgelegt wird. *Action* muß auf eine *lokale*, als **far** deklarierte Prozedur zeigen, die als Parameter einen Zeiger übernimmt:

```
procedure MyActionProc(P: PView); far;
```

Die im folgenden Ausschnitt gezeigte Methode *MoveSubViews* versetzt alle Subviews der Gruppe um den in *Delta* angegebenen Wert. Die Methoden *Lock* und *Unlock* werden dazu benutzt, die Zahl der Ausgabeoperationen zu beschränken und Bildschirmflimmern zu vermeiden:

```
procedure TMyGroup.MoveSubViews(Delta: TPoint);

  procedure DoMoveView(P: PView); far;
  begin
    P^.MoveTo(P^.Origin.X + Delta.X, P^.Origin.Y +
              Delta.Y);
  end;

begin
  Lock;
  ForEach(@DoMoveView);
  Unlock;
end;
```

Siehe auch: *TGroup.FirstThat*.

GetData

```
procedure GetData(var Rec); virtual;
```

Diese Methode ruft für jede Subview deren gleichnamige Methode auf, und zwar in der durch die Z-Ordnung festgelegten umgekehrten Reihenfolge. Siehe auch: *TView.GetData*, *TGroup.SetData*.

GetHelpCtx

```
function GetHelpCtx: Word; virtual;
```

Gibt den Wert des Feldes *HelpCtx* der gerade selektierten Subview zurück. Siehe auch: *TView.GetHelpCtx*.

GetSubViewPtr

```
procedure GetSubViewPtr(var S: TStream; var P);
```

Ermittelt einen Zeiger auf eine Subview, die im angegebenen Stream gespeichert ist. Diese Methode sollte nur innerhalb eines *Load*-Konstruktors aufgerufen werden, um solche Zeiger einzulesen, die innerhalb einer *Store*-Methode über *PutSubViewPtr* abgespeichert wurden. Siehe auch: *TGroup.Load*, *TGroup.PutSubViewPtr*, *TGroup.Store*.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Eine Gruppe bearbeitet Ereignisse, indem sie diese, abhängig von der Art des Ereignisses, an die selektierten Subviews weiterreicht.

Fokussierte Ereignisse (*evKeyDown* und *evCommand*) werden in drei Phasen bearbeitet. Zunächst wird im Feld *Phase* der Gruppe der Wert *phPreProcess* eingetragen und das Ereignis allen Subviews übergeben, in deren Feld *Options* das Bit *ofPreProcess* gesetzt ist. Darauf wird im Feld *Phase* der Wert *phFocused* eingetragen und das Ereignis dem *View*-Objekt übergeben, das die Eigenschaft *sfSelected* besitzt. Schließlich wird im Feld *Phase* der Wert *phPostProcess* eingetragen und das Ereignis allen Subviews übergeben, in deren Feld *Options* das Bit *ofPostProcess* gesetzt ist.

Positionsereignisse, in der Regel Ereignisse des Typs *evMouse*, werden dem ersten Objekt zur Bearbeitung übergeben, in dessen Bereich sich die in *Event.Where* angegebene Position befindet. Botschaften, also solche Ereignisse, die weder Positions- noch fokussierte Ereignisse sind, werden unter Beachtung der über die Z-Ordnung festgelegten Reihenfolge jeder Subview der Gruppe übergeben.

Hinweis: Wenn das Feld *EventMask* einer Subview die Bearbeitung bestimmter Ereignisarten ausschließt, werden Ereignisse dieses Typs **niemals** dieser Subview übergeben.

Siehe auch: *FocusedEvents*, *PositionalEvents*, *evXXXX*-Konstanten, *TView.EventMask*, *TView.HandleEvent*.

Insert

```
procedure Insert(P: PView);
```

Fügt *P^* als neue Subview in die Gruppe ein; kommt ihr die Eigenschaft *ofSelectable* zu, wird sie gleichzeitig zur aktuellen selektierten Subview. Siehe auch: *TGroup.First*, *TGroup.ExecView*, *TGroup.Delete*.

InsertBefore

```
procedure InsertBefore(P, Target: PView);
```

Fügt *P^* als neue Subview in die Gruppe ein, und zwar vor dem Objekt, auf das *Target* zeigt. Ist *Target* **nil**, wird das Objekt hinter allen anderen Subviews der Gruppe eingefügt. Siehe auch: *TGroup.Insert*, *TGroup.Delete*.

Lock

```
procedure Lock;
```

Besitzt die Gruppe einen Puffer, dann inkrementiert *Lock* das

private Feld *LockFlag*. Dadurch werden alle Ausgaben der Subviews solange in den Puffer umgeleitet, bis die Methode *Unlock* aufgerufen wird. Siehe auch: *TGroup.Draw*, *TGroup.Unlock*.

PutSubViewPtr

```
procedure PutSubViewPtr(var S: TStream; P: PView);
```

Schreibt einen Zeiger auf eine Subview in den angegebenen Stream. Diese Methode wird von *TGroup.Store* aufgerufen. Siehe auch: *TGroup.GetSubViewPtr*, *TGroup.Load*, *TGroup.Store*.

Redraw

```
procedure Redraw;
```

Veranlaßt unter Beachtung der über die Z-Ordnung festgelegten Reihenfolge die erneute Darstellung aller in der Gruppe zusammengefaßten Objekte. Im Unterschied zu *TGroup.Draw* greift *Redraw* nicht auf Puffer zu. Siehe auch: *TGroup.First*.

SelectNext

```
procedure SelectNext(Forwards: Boolean);
```

Ist *Forwards* True, macht diese Methode die in der durch die Z-Ordnung festgelegten Reihenfolge nächste Subview, die die Eigenschaft *ofSelectable* besitzt, zur aktuellen Subview. Ist *Forward* dagegen False, wird jene Subview mit der Eigenschaft *sfSelectable* zur aktuellen, die dem Objekt, auf das *Current* zeigt, vorangeht. Siehe auch: *ofXXXX*-Konstanten.

SetData

```
procedure SetData(var Rec); virtual;
```

Die Gruppe liest Daten aus *Rec* ein. Die Größe der Daten wird über einen Ausfruf von *DataSize* ermittelt. Mittels dieser Methode kann eine Gruppe über eine einzige Anweisung initialisiert werden. Siehe auch: *TGroup.DataSize*, *TGroup.GetData*.

SetState

```
procedure SetState(AState: Word; Enable: Boolean);
                   virtual;
```

Ruft für jede Subview deren gleichnamige Methode auf. Abhängig von *AState* und *Enable* werden darauf diese Objekte neu ausgegeben und der gerade selektierten Subview die Eigenschaft *sfFocused* zugeordnet. Siehe auch: *TView.SetState*.

Store

```
procedure Store(var S: TStream);
```

Schreibt alle in der Gruppe enthaltenen Objekte in den Stream. Siehe auch: *TView.Put*, *TView.Store*, *TGroup.PutSubViewPtr*, *TGroup.Load*.

Unlock

```
procedure Unlock;
```

Besitzt die Gruppe einen Puffer, dekrementiert *Unlock* das private Feld *LockFlag*. Erreicht dieses den Wert 0, wird die

DrawView-Methode aufgerufen, die den Pufferinhalt in den Bildschirmspeicher kopiert. Siehe auch: *TGroup.Lock*, *TView.DrawView*.

Valid

```
function Valid(Command: Word): Boolean; virtual;
```

Gibt True zurück, wenn die gleichnamigen Methoden aller Subviews der Gruppe ebenfalls True ermitteln. Über diese Methode kontrolliert *Execute*, ob Ereignisse erfolgreich abgearbeitet wurden. Eine Gruppe kann die Eigenschaft der Modalität nur dann abgeben, wenn *Valid* True zurückgibt. Siehe auch: *TView.HandleEvent*, *TView.Valid*, *TGroup.Execute*, *TGroup.HandleEvent*.

THistory

DIALOGS

Ein *THistory*-Objekt ist immer mit einem Objekt des Typs *TInputLine* verbunden und stellt eine Liste vorher getätigter Eingaben bereit. Der Benutzer kann einen Eintrag aus dieser Liste auswählen, was ihm die wiederholte Eingabe gleicher Werte erspart.

Ein *THistory*-Objekt macht sich rechts neben einem Eingabefeld durch einen nach unten weisenden Pfeil bemerkbar. Klickt der Benutzer dieses Symbol an, öffnet sich ein Fenster (zuständiges Objekt: *THistoryWindow*) mit einer Liste vorher getätigter Eingaben (zuständiges Objekt: *THistoryViewer*).

Felder

Link

```
Link: PInputLine;
```

Ein Zeiger auf das mit diesem Objekt verbundene Objekt des Typs *TInputLine*.

HistoryID

```
HistoryID: Word;
```

Jeder Eingabeliste ist eine Identifikationsnummer zugeordnet, die der Programmierer festlegt. Verschiedene Instanzen von *THistory* können, wenn ihnen die gleiche Nummer zugeordnet wurde, auf die gleiche Liste zugreifen.

Methoden

Init
```
constructor Init(var Bounds: TRect; ALink: PInputLine;
                 AHistoryId: Word);
```

Initialisiert über *TView.Init(Bounds)* ein *THistory*-Objekt in der in *Bounds* angegebenen Größe. Den Feldern *Link* und *HistoryID* werden *ALink* und *AHistoryID* zugewiesen. Im Feld *Options* wird das Bit *ofPostProcess* und in *EventMask* das Bit *evBroadcast* gesetzt. Siehe auch: *TView.Init*.

Load
```
constructor Load(var S: TStream);
```

Erstellt ein *THistory*-Objekt und initialisiert dessen Felder *Link* und *HistoryID* mit den Werten, die es über *TView.Load(S)* aus dem angegebenen Stream einliest. Siehe auch: *TView.Store*.

Draw
```
procedure Draw; virtual;
```

Gibt das *THistory*-Objekt in der über *GetPalette* ermittelten Farbe aus. Bei abgeleiteten Objekten muß diese Methode überschrieben werden.

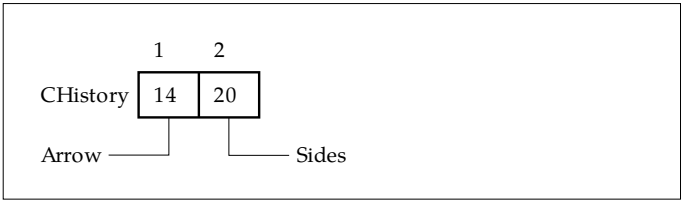
GetPalette
```
function GetPalette: PPalette; virtual;
```

Diese pseudo-abstrakte Methode gibt einen Zeiger auf die Farbpalette *CHistory* zurück.

Store
```
procedure Store(var S: TStream);
```

Schreibt über *TView.Store(S)* ein *THistory*-Objekt und dessen Felder *Link* und *HistoryID* in den angegebenen Stream. Siehe auch: *TView.Store*.

Farbpalette Dieses Objekt greift über die Palette *CHistory* auf die Einträge 14 bis 20 der Palette *CDialog* zu.



THistoryViewer DIALOGS

THistoryViewer ist ein Nachkomme von *TListViewer* und dafür zuständig, eine Liste von Eingabewerten innerhalb eines Fensters auszugeben, für dessen Darstellung wiederum das Objekt *THistoryWindow* verantwortlich ist.

Felder

HistoryID

```
HistoryID: Word;
```

Die Identifikationsnummer der darzustellenden Liste von Eingaben.

Methoden

Init

```
constructor Init(var Bounds: TRect; AHScrollBar,
                 AVScrollBar: PScrollBar; AHistoryId:
                 Word);
```

Initialisiert über *TListViewer.Init* ein *THistoryViewer*-Objekt in der in *Bounds* angegebenen Größe, versehen mit den übergebenen Rollbalken. Dem Feld *HistoryID* wird *AHistoryID* zugewiesen. Die dadurch bezeichnete Liste wird überprüft, um die Zahl und die Breite der Einträge zu erfahren. Der Fokus wird auf den ersten Eintrag dieser Liste gesetzt. Siehe auch: *TListViewer.Init*.

GetPalette

```
function GetPalette: PPalette; virtual;
```

Gibt einen Zeiger auf die Farbpalette *CHistoryViewer* zurück.

GetText

```
function GetText(Item: Integer; MaxLen: Integer):
                 String; virtual;
```

Gibt den Eintrag mit der Nummer *Item* aus der Liste zurück. *ListViewer.Draw* ruft diese Methode für jeden darzustellenden Eintrag auf. Siehe auch: *TListViewer.Draw*, *HistoryStr*.

HandleEvent

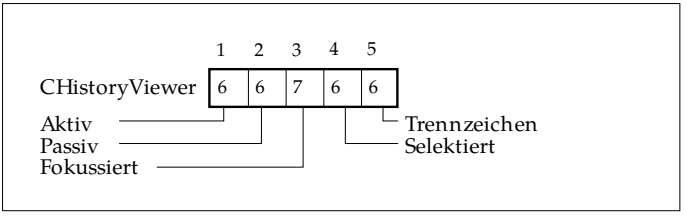
```
procedure HandleEvent(var Event: TEvent); virtual;
```

Selektiert der Benutzer einen Eintrag in der Liste mittels Doppelklick oder über [Enter], wird der Befehl *cmOk* abgesetzt. Drückt der Benutzer [Esc] oder klickt das Schließfeld des Fensters an, wird der Befehl *cmCancel* abgesetzt. Alle anderen Ereignisse werden an *TListViewer.HandleEvent* weitergeleitet. Siehe auch: *TListViewer.HandleEvent*.

HistoryWidth

```
function HistoryWidth: Integer;
```

Gibt die Länge des längsten Strings in der über *HistoryID* angesprochenen Liste zurück.



Farbpalette Dieses Objekt greift über die Palette *CHistoryViewer* auf die Einträge 6 und 7 der Palette *CDialog* zu.

THistoryWindow DIALOGS

THistoryWindow ist ein Nachkomme von *TWindow* und verwaltet das Fenster, innerhalb dessen ein *THistoryViewer*-Objekt vorher gespeicherte Eingabewerte darstellt. Dieses Fenster besitzt weder einen Titel, noch wird ihm eine Nummer zugewiesen. Es kann über ein Schließfeld geschlossen werden, jedoch weder seine Größe, noch seine Position ändern.

Felder

Viewer

```
Viewer: PListViewer;
```

Ein Zeiger auf das mit diesem Objekt verbundene Objekt des Typs *THistoryViewer*.

Methoden

Init

```
constructor Init(var Bounds: TRect; AHistoryId: Word);
```

Initialisiert über *TWindow.Init* ein *THistoryWindow*-Objekt in der in *Bounds* angegebenen Größe. Diesem Fenster wird weder ein Titel noch eine Nummer zugeordnet. In dem Feld *TWindow.Flags* ist das Bit *wfClose* gesetzt, um das Fenster mit einem Schließfeld zu versehen. Darauf wird das *TListViewer*-Objekt initialisiert, das verantwortlich für die Ausgabe der Liste ist. Siehe auch: *TWindow.Init, THistoryWindow.InitViewer*.

GetPalette

```
function GetPalette: PPalette; virtual;
```

Gibt einen Zeiger auf die Farbpalette *CHistoryWindow* zurück.

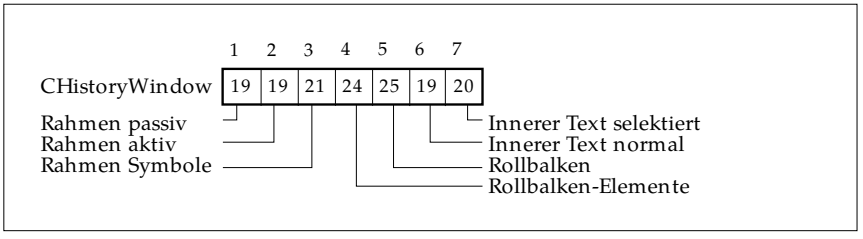
GetSelection

```
function GetSelection: String; virtual;
```

Gibt den String zurück, auf den das mit diesem Objekt verbundene *TListViewer*-Objekt den Fokus gelegt hat. Siehe auch: *THistoryViewer.GetText*.

InitViewer

```
procedure InitViewer(HistoryID: Word); virtual;
```

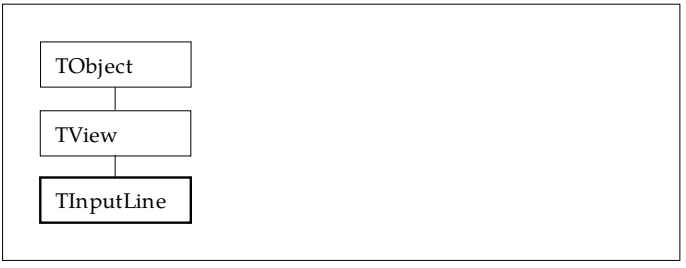


Instantiiert ein *THistoryViewer*-Objekt, das verantwortlich ist für die Darstellung der Einträge in der Liste, die über *HistoryID* identifiziert wird. Siehe auch: *THistoryViewer.Init*.

Farbpalette Dieses Objekt greift über die Palette *CHistoryWindow* auf die Einträge 19 bis 25 der Palette *CDialog* zu.

TInputLine

DIALOGS



TInputLine stellt einen Zeileneditor zur Verfügung, der die gewohnten Editiertasten und die Maus unterstützt. Mittels der Maus kann ein Textblock innerhalb der bearbeiteten Zeile markiert werden. Diese Zeile kann länger als der am Bildschirm zur Verfügung stehende Raum sein, da sie bei Bedarf horizontal gerollt wird. Über die Methode *SetData* nimmt dieses Objekt die zu bearbeitenden Daten entgegen und gibt sie über *GetData* wieder zurück.

Felder

Data `Data: PString;`

Zeiger auf den zu bearbeitenden String.

MaxLen `MaxLen: Integer;`

Maximale Länge, die der String erreichen darf (ohne das Längenbyte). Siehe auch: *TInputLine.DataSize*.

CurPos `CurPos: Integer;`

Index auf die aktuelle Position innerhalb des Strings. Siehe auch: *TInputLine.SelectAll*.

FirstPos `FirstPos: Integer;`

Index auf das erste dargestellte Zeichen des Strings. Siehe auch: *TInputLine.SelectAll*.

SelStart `SelStart: Integer;`

Index auf das erste Zeichen eines markierten Blocks. Siehe auch: *TInputLine.SelectAll*.

SelEnd

```
SelEnd: Integer;
```

Index auf das letzte Zeichen eines markierten Blocks. Siehe auch: *TInputLine.SelectAll*.

Methoden

Init

```
constructor Init(var Bounds: TRect; AMaxLen: Integer);
```

Initialisiert über *TView.Init*(*Bounds*) ein Eingabefeld in der in *Bounds* angegebenen Größe. In *Options* werden die Bits *ofSelectable* und *ofFirstClick*, in *State* das Bit *sfCursorVis* gesetzt. Über *GetMem* werden *AMaxLen* + 1 Bytes Speicher reserviert und initialisiert. Das Feld *Data* zeigt auf diesen Speicherbereich. Siehe auch: *TView.Init*.

Load

```
constructor Load(var S: TStream);
```

Erstellt ein *TInputLine*-Objekt und initialisiert es mittels *TView.Load*(*S*) mit Werten, die es über *S.Read* aus dem Stream einliest. Mittels *GetMem* werden *MaxLen* + 1 Bytes Speicher reserviert. Diese Methode wird zusammen mit *TInputLine.Store* genutzt, um Objekte des Typs *TInputLine* in einem Stream zu speichern bzw. wieder zu laden. Siehe auch: *TView.Load*, *TInputLine.Store*, *TStream.Read*.

Done

```
destructor Done; virtual;
```

Gibt den mit *Data* verbundenen Speicher frei und ruft dann den Destruktor *TView.Done* auf, der das Objekt entfernt. Siehe auch: *TView.Done*.

DataSize

```
function DataSize: Word; virtual;
```

Diese pseudo-abstrakte Methode gibt die Größe des Datenblocks zurück, den die Methoden *GetData* und *SetData* aus bzw. in den Datenrecord kopieren. Sofern nicht überschrieben, gibt diese Methode den Wert *MaxLen* + 1 zurück. Siehe auch: *TInputLine.GetData*, *TInputLine.SetData*.

Draw

```
procedure Draw; virtual;
```

Stellt das Eingabfeld auf dem Bildschirm dar und verwendet dabei die Farben, die dem Status des Objekts entsprechen. Pfeilsymbole werden dann ausgegeben, wenn die Eingabezeile in ein oder zwei Richtungen horizontal gerollt werden kann. Ist ein Bereich dieser Zeile markiert, wird auch dieser Block mit der entsprechenden Farbe markiert.

GetData

```
procedure GetData(var Rec); virtual;
```

Kopiert *DataSize* Bytes Daten aus dem String *Data*^ nach *Rec*. Diese Methode übernimmt zusammen mit *SetData* den Austausch der zu bearbeitenden Daten zwischen diesem und

anderen Objekten. Siehe auch: *TInputLine.DataSize*, *TInputLine.SetData*.

GetPalette

```
function GetPalette: PPalette; virtual;
```

Diese pseudo-abstrakte Methode gibt einen Zeiger auf die Farbpalette *CInputLine* zurück.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Ruft zunächst *TView.HandleEvent* auf und kümmert sich dann um alle Tastatur- und Mausereignisse, die den Zeileneditor betreffen. Diese Methode stellt damit die Editier-Eigenschaften dieses Objekts bereit: Blockmarkierung mittels der Maus, Löschen dieses Blocks, horizontales Rollen, wenn erforderlich, Umschaltung zwischen Einfüge- und Überschreibmodus, wobei sich die Form des Cursors ändert, Unterstützung der Cursortasten, einschließlich [Home] und [End], wobei diese Funktionen auch über die Kombination [Ctrl][Taste] erreichbar sind. *HandleEvent* veranlaßt auch die Neudarstellung der Eingabezeile. Siehe auch: *sfCursorIns*, *TView.HandleEvent*, *TInputLine.SelectAll*.

SelectAll

```
procedure SelectAll(Enable: Boolean);
```

Setzt die Felder *CurPos*, *FirstPos* und *SelStart* auf den Wert 0. Ist *Enable* True, erhält das Feld *SelEnd* den Wert *Length*(*Data*^), wodurch die gesamte Eingabezeile markiert wird. Ist jedoch *Enable* False, wird auch *SelEnd* auf 0 gesetzt. Danach wird *DrawView* aufgerufen, um das Objekt erneut darzustellen. Siehe auch: *TView.DrawView*.

SetData

```
procedure SetData(var Rec); virtual;
```

Kopiert *DataSize* Bytes Daten aus *Rec* in den String *Data*^ und ruft dann *SelectAll*(*True*) auf, damit die übrigen Datenfelder initialisiert werden. Diese Methode übernimmt zusammen mit *GetData* den Austausch der zu bearbeitenden Daten zwischen diesem und anderen Objekten. Siehe auch: *TInputLine.DataSize*, *TInputLine.GetData*, *TView.DrawView*.

SetState

```
procedure SetState(AState: Word; Enable: Boolean);
                   virtual;
```

Ruft *TView.SetState*(*AState, Enable*) auf, damit – sofern *Enable* True ist – alle in *AState* gesetzten Bits auch in *State* gesetzt oder – wenn *Enable* False ist – gelöscht werden. Sodann wird, wenn die Bits *sfSelected* oder *sfActive* in *AState* gesetzt sind, *SelectAll*(*Enable*) aufgerufen. Siehe auch: *TView.SetState*, *TInputLine.SelectAll*.

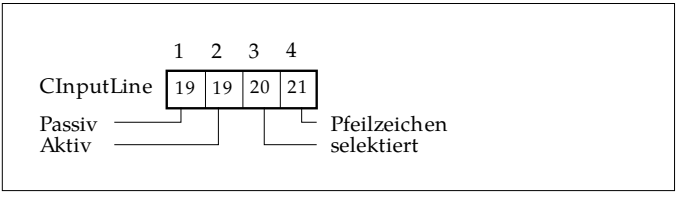
Store

```
procedure Store(var S: TStream);
```

Schreibt ein *TInputLine*-Objekt in den angegebenen Stream,

indem es über *TView.Store(S)*, dem Aufrufe von *S.Write* folgen, sowohl den Wert von *MaxLen* als auch den String *Data^* dem Stream übergibt. Diese Methode wird zusammen mit *TInputLine.Load* genutzt, um Objekte des Typs *TInputLine* in einem Stream zu speichern bzw. wieder zu laden. Siehe auch: *TView.Store*, *TInputLine.Load*, *TStream.Write*.

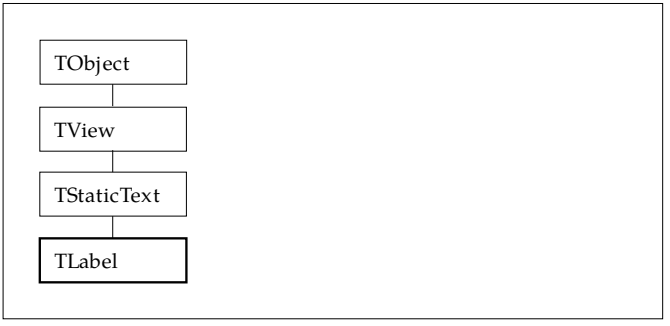
Farbpalette Dieses Objekt greift über die Palette *CInputLine* auf die Einträge 19 bis 21 der Palette *CDialog* zu.



TLabel

DIALOGS

TLabel ist ein Nachkomme von *TStaticText*, unterscheidet



sich von diesem aber dadurch, daß ein Label mittels der Maus, über die Cursortasten oder auch über eine Tastenkombination aus `[Alt][Buchstabe]` selektiert werden kann. Normalerweise ist ein Label über einen *PView*-Zeiger mit einem anderen Dialogelement, einem Eingabefeld, einem Auswahlfenster oder einem Schaltfeld, verbunden. Wird das Label selektiert, wird auch das verbundene Element selektiert. Wird umgekehrt dieses Dialogelement selektiert, wird auch das Label hervorgehoben dargestellt.

Felder

Link

```
Link: PView;
```

Ein Zeiger auf das mit diesem Objekt verbundene Dialogelement.

Light `Light: Boolean;`

Wenn True, dann ist das Label und das damit verbundene Dialogelement selektiert; beide müssen hervorgehoben dargestellt werden.

Methoden

Init
```
constructor Init(var Bounds: TRect; AText: String;
                 ALink: PView);
```

Initialisiert über *TStaticText.Init*(*Bounds*) ein *TLabel*-Objekt in der in *Bounds* angegebenen Größe. Im *Options*-Feld werden die Bits *ofPreProcess* und *ofPostProcess* gesetzt und *EventMask* wird auf *evBroadCast* gesetzt. Dem Feld *Link* wird *ALink* zugewiesen, ein Zeiger auf das mit dem Label verbundene Dialogelement. Übergeben Sie in *ALink* den Wert **nil**, wenn eine solche Anbindung nicht notwendig ist. Siehe auch: *TStaticText.Init.*

Load `constructor Load(var S: TStream);`

Lädt ein *TLabel*-Objekt aus dem angegebenen Stream. Siehe auch: *TLabel.Store.*

Draw `procedure Draw; virtual;`

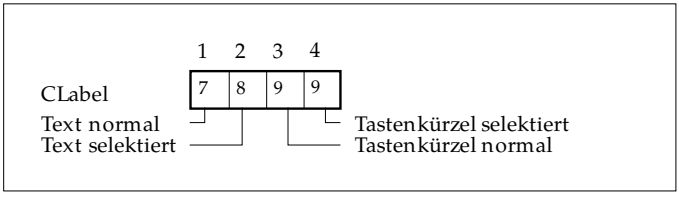
Gibt in der Farbe, die dem Status des Objekts entspricht, den String *Text*^ aus. Siehe auch: *TStaticText.Text.*

GetPalette `function GetPalette: PPalette; virtual;`

Diese pseudo-abstrakte Methode gibt einen Zeiger auf die Farbpalette *CLabel* zurück.

HandleEvent `procedure HandleEvent(var Event: TEvent); virtual;`

Ruft die Methode *TView.HandleEvent* auf. Wenn das Ereignis *evMouseDown* auftrat oder das für dieses Objekt definierte Tastenkürzel betätigt wurde, wird – sofern vorhanden – das verbundene Dialogelement selektiert. Diese Methode ist auch zuständig für von diesem Dialogelement ausgehende Ereignisse des Typs *evBroadCast*, wobei, wenn notwendig, in *Light* der zutreffende Wert eingetragen und das Label neu ausgegeben wird. Siehe auch: *TView.HandleEvent*, *cmXXXX*-Konstanten.



Store

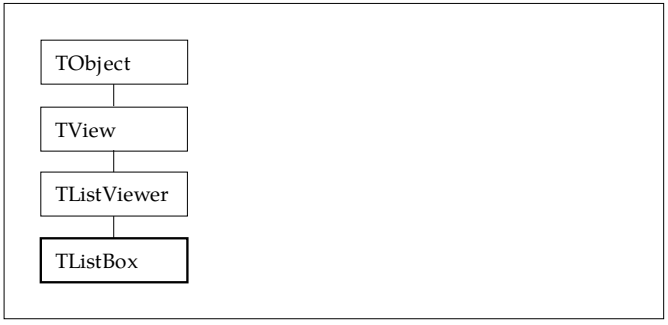
```
procedure Store(var S: TStream); virtual;
```

Speichert ein *TLabel*-Objekt im angegebenen Stream. Siehe auch: *TLabel.Load*.

Farbpalette Dieses Objekt greift über die Palette *CLabel* auf die Einträge 7 bis 9 der Palette *CDialog* zu.

TListBox

Dialogs



TListBox ist ein von *TListViewer* abgeleitetes Objekt, das Ihnen ermöglicht, Listen miteinander verketteter Strings, zum Beispiel eine Liste von Dateinamen, auszugeben. Eine solche Liste kann in einer oder mehreren Spalten innerhalb des Darstellungsbereichs ausgegeben werden, wobei auch ein vertikaler Rollbalken, nicht jedoch ein horizontaler Rollbalken unterstützt wird. Die von *TListViewer* geerbten Methoden erlauben die Selektion eines Eintrags der Liste mittels der Maus oder über die Cursortasten. Dieses Objekt überschreibt weder die geerbte *HandleEvent*- noch die *Draw*-Methode, definiert aber das zusätzliches Feld *List*, das einen Zeiger auf ein *TCollection*-Objekt, das die darzustellenden Einträge bereithält, speichert. Was diese Kollektion enthält, liegt in Ihrer Verantwortung, und das gilt auch für die Aktion, die ausgelöst wird, wenn ein Eintrag der Liste selektiert ist.

Felder

List

```
List: PCollection;
```

Zeigt auf eine Kollektion, die die darzustellenden Einträge enthält; diese Einträge sind in der Regel vom Typ *PString*.

Methoden

Init

```
constructor Init(var Bounds: TRect; ANumCols: Word; AS
                 crollBar: PScrollBar);
```

Erstellt ein *TListBox*-Objekt von der in *Bounds* angegebenen

Größe, mit *ANumCols* Spalten und einem über *AScrollBar* referenzierten Rollbalken. Dieser Konstruktor ruft *TListViewer.Init* auf (und übergibt **nil** statt eines Zeigers auf einen horizontalen Rollbalken). Das Feld *List* wird auf **nil** gesetzt und Range auf 0. Die Liste ist also zunächst leer. Ihr Programm muß ein *TCollection*-Objekt, das die darzustellenden Einträge speichert, bereitstellen und einen Zeiger darauf in List ablegen. Siehe auch: *TListViewer.Init, TListBox.SetData*.

Load

```
constructor Load(var S: TStream);
```

Erstellt ein *TListBox*-Objekt und initialsiert es mit den aus dem angegebenen Stream gelesenen Werten. Dazu ruft dieser Konstruktor *TListViewer.Load*(*S*) auf und liest über *S.Get* den *List*-Zeiger ein. Siehe auch: *TListViewer.Load*, *TStream.Get*, *TListBox.Store*.

DataSize

```
function DataSize: Word; virtual;
```

Gibt die Größe der Daten zurück, die über *SetData* eingelesen und über *GetData* zurückgegeben werden. Sofern nicht überschrieben, gibt diese Methode die Größe des intern verwendeten *TListBoxRec*-Records zurück, der einen *List*-Zeiger und einen Index auf den gerade fokussierten Eintrag speichert. Verwenden Sie in abgeleiteten Objekten andere Daten, müssen Sie diese Methode überschreiben. Siehe auch: *TListBox.GetData*, *TListBox.SetData*.

GetData

```
procedure GetData(var Rec); virtual;
```

Kopiert *DataSize* Bytes Daten in *Rec*. Sofern nicht überschrieben, übergibt diese Methode den *List*-Zeiger und die Nummer des gerade fokussierten Eintrags, also den Inhalt des Feldes *Focused*. Verwenden Sie in abgeleiteten Objekten andere Daten, müssen Sie diese Methode überschreiben. Siehe auch: *TListBox.DataSize*, *TListBox.SetData*.

GetText

```
function GetText(Item: Integer; MaxLen: Integer):
                 String; virtual;
```

Gibt über die Anweisung *PString*(*List^.At*(*Item*))^ den String mit der Nummer *Item* aus der *TCollection* zurück oder einen leeren String, wenn *List* **nil** ist. Sind die Einträge in Ihrer Liste nicht vom Typ String, müssen Sie diese Methode überschreiben. Siehe auch: *TCollection.At.*

NewList

```
procedure NewList(AList: PCollection); virtual;
```

Ist *AList* ungleich **nil**, ersetzt die über AList referenzierte Datenliste die aktuelle Liste, auf die *List* zeigt. Das Feld Range wird auf den Wert des Feldes *Count* der neuen *TCollection* gesetzt, deren erster Eintrag über *FocusItem*(*0*) fokussiert wird. Schließlich wird durch einen Aufruf von *DrawView* die

Darstellung der Liste veranlaßt. Beachten Sie, daß der von der alten Liste eventuell belegte Speicher freigegeben wird, bevor die neuen Werte übernommen werden. Siehe auch: *TListBox.SetData*, *TListViewer.SetRange*, *TListViewer.FocusItem*, *TView.DrawView*.

SetData

```
procedure SetData(var Rec); virtual;
```

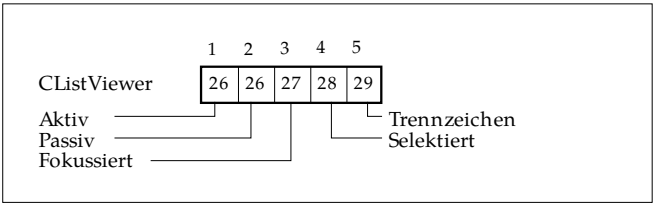
Sofern nicht überschrieben, übernimmt diese einen *List*-Zeiger und die Nummer des fokussierten Eintrags (der im Feld *Focused* abgelegt wird) aus *Rec*. Danach wird *NewList* aufgerufen; diese Methode veranlaßt die Ausgabe der neuen Liste und die Markierung des fokussierten Eintrags. Wie auch die Methoden *GetData* und *DataSize*, müssen Sie diese Methode in abgeleiteten Objekten überschreiben. Siehe auch: *TListBox.DataSize*, *TListBox.GetData*, *TListBox.NewList*.

Store

```
procedure Store(var S: TStream);
```

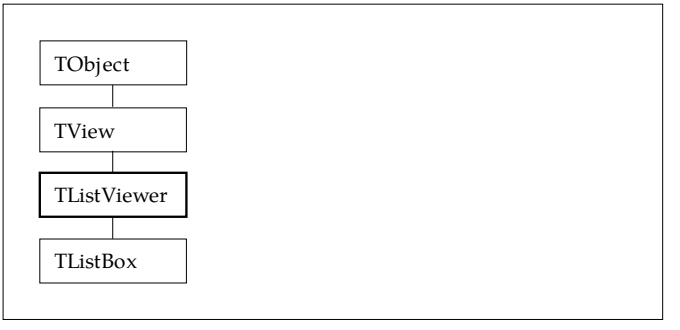
Schreibt über *TListViewer.Store(S)* ein *TListBox*-Objekt, dann mittels *S.Put* das Feld *List* in den angegebenen Stream. Siehe auch: *TListBox.Load*, *TListViewer.Store*, *TStream.Put*.

Farbpalette Dieses Objekt greift über die Palette *CListViewer* auf die Einträge 26 bis 29 der aktuellen Farbpalette des Programms zu.



TListViewer

VIEWS



TListViewer ist der ideale Vorfahre solcher Objekte, deren Aufgabe darin besteht, den Inhalt von Datenlisten auszugeben. Die Felder und Methoden, die *TListViewer* definiert, stellen bereits die grundlegende Funktionalität solcher Objekte bereit:

- Es bietet ein *View*-Objekt, das verkettete Listen von Dateneinträgen darstellen kann.
- Es steuert die Darstellung über ein oder zwei Rollbalken.
- Es bietet die Möglichkeit, ein »Fenster« in zwei Richtungen über einen wesentlichen größeren Datenbereich zu bewegen.
- Dieses Objekt kann zusammen mit den Rollbalken in einen Stream gespeichert und daraus wieder gelesen werden.
- Einträge der Liste können mittels der Maus oder über die Tastatur selektiert werden.
- Die Methode *Draw* kommt auch mit einer Größenänderung des Darstellungsbereichs zurecht.

TListViewer definiert eine abstrakte Methode *GetText*, die dazu gedacht ist, daß das Objekt die darzustellenden Daten anfordern kann. Sie müssen diese Methode überschreiben, damit die Daten in der von Ihnen gewünschten Form ausgegeben werden können.

Felder

HScrollBar `HScrollBar: PScrollBar;`

Zeigt auf den mit diesem Objekt verbundenen horizontalen Rollbalken oder enthält den Wert **nil**, wenn kein Rollbalken definiert ist.

VScrollBar `VScrollBar: PScrollBar;`

Zeigt auf den mit diesem Objekt verbundenen verikalen Rollbalken oder enthält den Wert **nil**, wenn kein Rollbalken definiert ist.

NumCols `NumCols: Integer;`

Zahl der Spalten, in die der Darstellungsbereich aufgeteilt wird.

TopItem `TopItem: Integer;`

Die Nummer des ersten dargestellten Eintrags; Einträge werden von 0 bis *Range* – 1 durchnumeriert. Dieser Wert hängt ab von der Zahl der Spalten und der Größe des Darstellungsbereichs. Siehe auch: *Range*.

Focused `Focused: Integer;`

Die Nummer des fokussierten Eintrags; die Einträge werden von 0 bis *Range* –1 durchnumeriert. Der Fokus kann mittels eines Mausklicks oder über [Tab] auf einen anderen Eintrag gelegt werden. Siehe auch: *Range*.

Range

```
Range: Integer;
```

Die aktuelle Gesamtzahl der Einträge in der Liste; sie werden von 0 bis *Range* – 1 durchnumeriert. Siehe auch: *TListViewer.SetRange*.

Methoden

Init

```
constructor Init(var Bounds: TRect; ANumCols: Word;
                 AHScrollBar, AVScrollBar: PScrollBar;);
```

Initialisiert über *TGroup.Init*(*Bounds*) ein *TListViewer*-Objekt in der in *Bounds* angegebenen Größe. *ANumCols* wird dem Feld *NumCols* zugewiesen. In *Options* werden die Bits *ofFirstClick* und *ofSelectable* gesetzt, so daß Mausklicks, die dieses Objekt selektieren, zunächst von dessen *HandleEvent*-Methode bearbeitet und dann erst weitergereicht werden. *EventMask* wird auf *evBroadCast*, die Felder *Range* und *Focused* auf 0 gesetzt. *AHScrollBar* und *AVScrollBar* sind Zeiger auf vertikale und horizontale Rollbalken. Sind diese oder einer von ihnen nicht notwendig, müssen Sie **nil** übergeben. Die beiden Zeiger werden in die Felder *HScrollBar* und *VScrollBar* eingetragen. Wenn Sie solche Rollbalken bereitgestellt haben, werden deren Felder *PgStep* und *ArStep* auf Werte gesetzt, die der Größe und der Zahl der Spalten des *TListViewer*-Objekts entsprechen. Definiert dieses Objekt beispielsweise nur eine Spalte, dann enthält das Feld *PgStep* des vertikalen Rollbalkens den Wert Size.Y – 1 und das Feld *ArStep* dieses Rollbalkens den Wert 1. Siehe auch: *TView.Init*, *TScrollBar.SetStep*.

Load

```
constructor Load(var S: TStream);
```

Erstellt ein *THistory*-Objekt und initialisiert es über *TView.Load*(*S*). Der Wert von *NumCols* wird aus dem Stream gelesen, ebenso die Rollbalken, wenn diese definiert sind. Siehe auch: *TView.Load*, *TListViewer.Store*.

ChangeBounds

```
procedure ChangeBounds(var Bounds: TRect); virtual;
```

Ändert über einen Aufruf von *TView.ChangeBounds*(*Bounds*) die Größe des *TListViewer*-Objekts. Ist mit diesem ein horizontaler Rollbalken verbunden, wird auch dessen Feld *PgStep* mit einem passenden Wert belegt. Siehe auch: *TView.ChangeBounds*, *TScrollBar.ChangeStep*.

Draw

```
procedure Draw; virtual;
```

Stellt das Objekt dar, wobei diese Methode auch berücksich-

tigt, ob *sfActive* gesetzt und welcher Eintrag fokussiert und selektiert ist.

FocusItem

```
procedure FocusItem(Item: Integer); virtual;
```

Setzt das Feld *Focused* auf *Item* und fokussiert dadurch diesen Eintrag. Diese Methode setzt auch, wenn ein vertikaler Rollbalken definiert ist, dessen Feld *Value* auf *Item* und belegt *TopItem* mit dem zutreffenden Wert. Siehe auch: *TListViewer.IsSelected*, *TScrollBar.SetValue*.

GetPalette

```
function GetPalette: PPalette; virtual;
```

Diese pseudo-abstrakte Methode gibt einen Zeiger auf die Farbpalette *CListViewer* zurück.

GetText

```
function GetText(Item: Integer; MaxLen: Integer):
                 String; virtual;
```

Diese abstrakte Methode müssen abgeleitete Objekte so überschreiben, daß *GetText* einen Eintrag *Item* des Typs String zurückgibt, dessen Länge den Wert *MaxLen* nicht überschreitet.

IsSelected

```
function IsSelected(Item: Integer): Boolean; virtual;
```

Gibt True zurück, wenn der Eintrag *Item* fokussiert ist, wenn also gilt *Focused* = *Item*. Siehe auch: *TlistViewer.FocusItem*.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Bearbeitet Ereignisse über einen Aufruf von *TView.HandleEvent*. Mittels Mausklick oder über die Cursortasten und `[PgDn]`, `[PgUp]`, `[Ctrl][PgDn]`, `[Ctrl][PgUp]`, `[Home]`, `[End]` bewegt der Benutzer den Fokus über die Einträge der Liste. Ein doppelter Mausklick oder `[Leertaste]` selektiert den gerade fokussierten Eintrag. Schließlich werden auch von den Rollbalken ausgesendete Ereignisse des Typs *evBroadCast* abgearbeitet, wobei, wenn notwendig, der Fokus auf einen anderen Eintrag versetzt und das Objekt neu dargestellt wird. Siehe auch: *TView.HandleEvent*, *TListViewer.FocusItem*.

SelectItem

```
procedure SelectItem(Item: Integer); virtual;
```

Sofern nicht überschrieben, sendet diese Methode über die Anweisung:

```
Message(Owner, evBroadCast, cmListItemSelected, @Self);
```

den Befehl *cmListItemSelected* an die Gruppe, zu der das Objekt gehört. Siehe auch: *TListViewer.FocusItem*.

SetRange

```
procedure SetRange(ARange: Integer);
```

Setzt das Feld *Range* auf den Wert *ARange*. Ist ein vertikaler Rollbalken definiert, werden auch die Felder dieses Objekts

angepaßt. Ist die Nummer des aktuell fokussierten Eintrags außerhalb des Bereichs von *Range*, wird im Feld *Focused* 0 eingetragen. Siehe auch: *TListViewer*, *Range*, *TScrollBar.SetParams*.

SetState

```
procedure SetState(AState: Word; Enable: Boolean);
                   virtual;
```

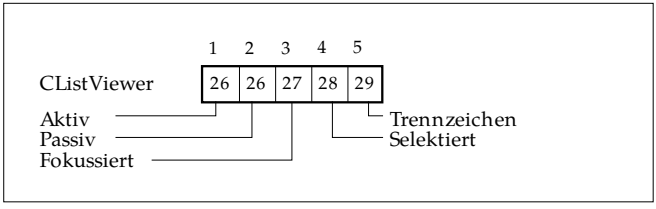
Ruft *TView.SetState*(*AState, Enable*) auf, um den Status des *TListViewer*-Objekts gemäß der in *AState* angegebenen Weise zu ändern. Dies kann dazu führen, daß es vom Bildschirm genommen oder neu dargestellt wird. Sind die Bits *sfSelected* und *sfActive* in *AState* gesetzt, werden die Rollbalken neu ausgegeben; ist in *AState* das Bit *sfSelected* gesetzt, aber das Bit *sfActive* gelöscht, werden die Rollbalken vom Bildschirm entfernt. Siehe auch: *TView.SetState*, *TScrollBar.Show*, *TScrollBar.Hide*.

Store

```
procedure Store(var S: TStream);
```

Ruft *TView.Store*(*S*) auf, um das Objekt in dem angegebenen Stream zu speichern, und übergibt dann diesem Stream auch die Rollbalken, sofern sie definiert sind. Siehe auch: *TView.Store*, *TListViewer.Load*.

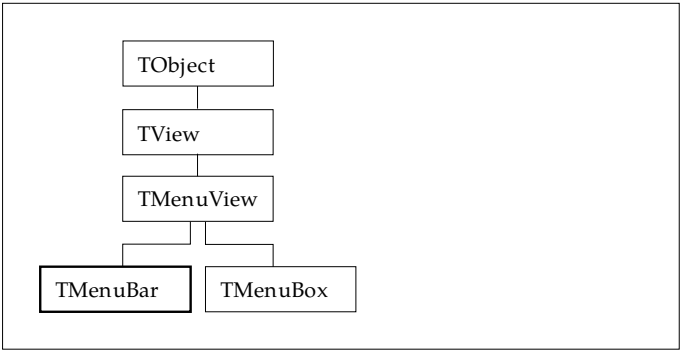
Farbpalette Dieses Objekt greift über die Palette *CListViewer* auf die Einträge 26 bis 29 der aktuellen Farbpalette des Programms zu.



TMenuBar MENUS

TMenuBar stellt eine Menüleiste bereit, deren einzelne Einträge mit einer der folgenden Methoden selektierbar sind:

- Sie klicken mit der Maus die gewünschte Option an;
- Sie aktivieren mit [F10] die Menüleiste und drücken dann die Taste, die dem hervorgehobenen Buchstaben der Option entspricht;
- Sie versetzen mit den Cursortasten den Balkencursor auf die Option und drücken [Enter] ;



- Sie bedienen sich der Tastenkombination [Alt] plus der Taste, die dem hervorgehobenen Buchstaben der Option entspricht.

Eine Menüleiste gehört normalerweise zu der Gruppe *TApplication* und stellt die zentralen Optionen des Programms zur Wahl. Mit diesen Optionen können *TMenuBox*-Objekte verbunden werden, die weitere Menüoptionen auflisten.

In einem Turbo-Vision-Programm haben Sie es in der Regel nicht direkt mit solchen Menü-Objekten zu tun. Sie müssen lediglich die Methode *TApplikation.InitMenuBar* überschreiben und in dieser neuen Methode die Routinen *NewMenu*, *NewSubMenu*, *NewItem* und *NewLine* aufrufen, und Turbo Vision erledigt den Aufbau dieser Menüs.

Methoden

Init

```
constructor Init(var Bounds: TRect; AMenu: PMenu);
```

Initialisiert über *TMenuView.Init*(*Bounds*) eine Menüleiste in der in *Bounds* angegebenen Größe. In *Options* wird das Bit *ofPreProcess*, in *GrowMode* das Bit *gfGrowHiX* gesetzt. Dem Feld *Menu* wird *AMenu* zugewiesen, wodurch die über dieses Menü wählbaren Optionen festgelegt sind. Siehe auch: *TMenuView.Init*, *TMenuView.Menu*.

Draw

```
procedure Draw; virtual;
```

Stellt das Menü auf dem Bildschirm dar. Die Felder *Name* und *Disabled* eines jeden *TMenuItem*-Records in der Liste werden überprüft, damit die Menüoption in den Farben ausgegeben wird, die ihrem Status entspricht. Die aktuelle Option wird hervorgehoben dargestellt.

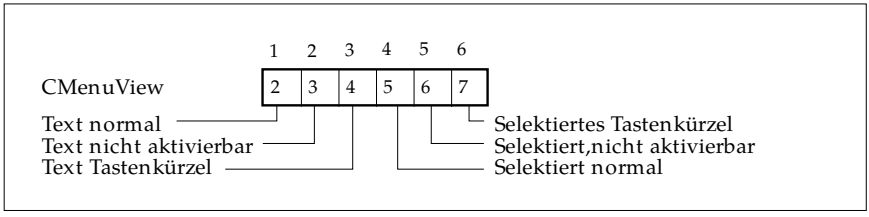
GetItemRect

```
procedure GetItemRect(Item: PMenuItem; var R: TRect);
                                        virtual;
```

Diese Methode überschreibt die gleichnamige Methode von *TMenuView*. Sie gibt die Begrenzungen des von *Item* belegten

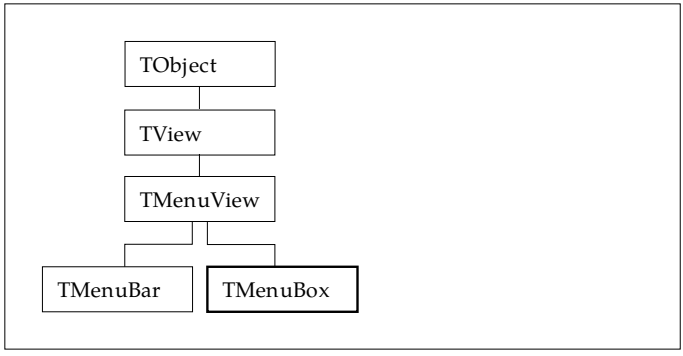
Bereichs zurück; dadurch ist es möglich zu ermitteln, ob dieser Bereich mit der Maus angeklickt wurde. Siehe auch: *TMenuView.GetItemRect*, *TMenuBox.GetItemRect*.

Farbpalette Dieses Objekt greift über die Palette *CMenuView* auf die Einträge 2 bis 7 der aktuellen Farbpalette des Programms zu.



TMenuBox

MENUS



Dieses Objekt stellt Menüfenster zur Verfügung, die eine Liste möglicher Optionen zur Wahl stellen. Diese Optionen können wiederum ein solches Menüfenster aktivieren. Wie bei einer Menüleiste, wird durch unterschiedliche Farbgebung angezeigt, welche Optionen deaktiviert sind und welche der Benutzer gerade selektieren kann.

Methoden

Init

```
constructor Init(var Bounds: TRect; AMenu: PMenu;
                 AParentMenu: PMenuView);
```

Initialisiert über *TMenuView.Init*(*Bounds*) ein Menüfenster in der in *Bounds* angegebenen Größe, die gegebenenfalls an die Zahl der verfügbaren Optionen angepaßt wird. In *Options* wird das Bit *ofPreProcess* gesetzt. Dem Feld *Menu* wird *AMenu* zugewiesen, wodurch die über dieses Menü wählbaren Optionen festgelegt sind. *State* wird auf *ofShadow* und das

Feld *ParentMenu* auf *AParentMenu* gesetzt. Siehe auch: *TMenuView.Init*, *TMenuView.Menu*.

Draw

```
procedure Draw; virtual;
```

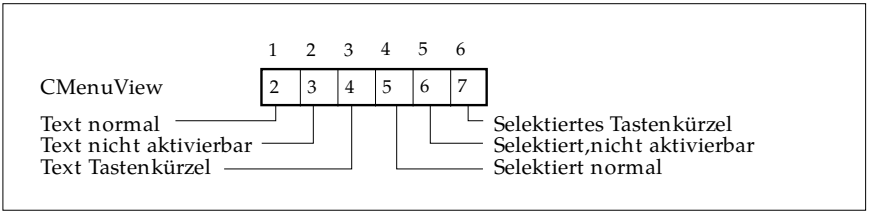
Stellt das umrahmte Menü auf dem Bildschirm dar. Die Felder *Name* und *Disabled* eines jeden *TMenuItem*-Records in der Liste werden überprüft, damit die Menüoption in den Farben ausgegeben wird, die ihrem Status entspricht. Die aktuelle Option wird hervorgehoben dargestellt.

GetItemRect

```
procedure GetItemRect(Item: PMenuItem; var R: TRect);
                                        virtual;
```

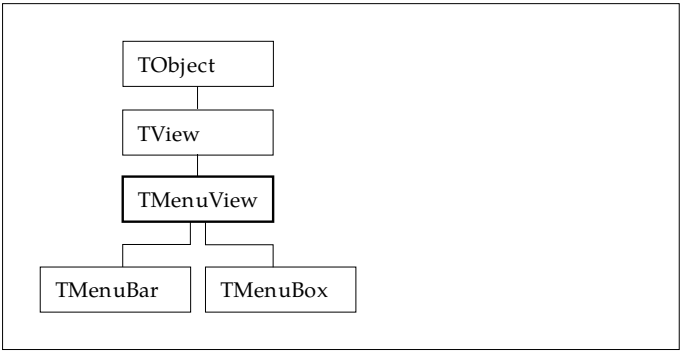
Diese Methode überschreibt die gleichnamige Methode von *TMenuView*. Sie gibt die Begrenzungen des von *Item* belegten Bereichs zurück; dadurch ist es möglich zu ermitteln, ob dieser Bereich mit der Maus angeklickt wurde. Siehe auch: *TMenuView.GetItemRect*, *TMenuBar.GetItemRect*.

Farbpalette Dieses Objekt greift über die Palette *CMenuView* auf die Einträge 2 bis 7 der aktuellen Farbpalette des Programms zu.



TMenuView

MENUS



TMenuView ist ein abstrakter Objekttyp und gemeinsamer Vorfahre der Objekte *TMenuBar* und *TMenuBox*, die Menüleisten und Menüfenster zur Verfügung stellen.

Felder

ParentMenu `ParentMenu: PMenuView;`

Ein Zeiger auf ein *TMenuView*- oder davon abgeleitetes Objekt, mit dem dieses Menü verbunden ist. Beachten Sie, daß *TMenuView* keine Gruppe ist. Solche Objekte sind miteinander verkettet, so daß ein Untermenü selektiert und dann wieder zum übergeordneten Menü zurückgekehrt werden kann. Siehe auch: *TMenuBox.Init*.

Menu `Menu: PMenu;`

Ein Zeiger auf einen Record des Typs *TMenu*, der den Zugriff auf alle in diesem Menü enthaltenen Menüoptionen ermöglicht. Siehe auch: *TMenuView.FindItem*, *TMenuView.GetItemRect*, *TMenuView.HotKey*, *TMenuBox.Init*.

Current `Current: PMenuItem;`

Ein Zeiger auf die gerade selektierte Menüoption. Siehe auch: *TMenuView.MenuSelect*.

Methoden

Init `constructor Init(var Bounds: TRect);`

Initialisiert über *TView.Init*(*Bounds*) ein *TMenuView*-Objekt in der in *Bounds* angegebenen Größe, dessen Feld *EventMask* auf den Wert *evBroadCast* gesetzt wird. Siehe auch: *TView.Init*, *evBroadCast*, *TApplication.InitMenuBar*.

Load `constructor Load(var S : TStream);`

Lädt ein *TMenuView*-Objekt aus dem angegebenen Stream. Siehe auch: *TView.Load*, *TMenuView.Store*.

Execute `function Execute: Word; virtual;`

Gibt das Menü aus und erlaubt dem Benutzer, eine Option auszuwählen. Die Funktion gibt den mit der Menüoption verbundenen Befehl zurück oder 0, wenn der Benutzer die Auswahl abgebrochen hat. Diese Methode darf nur von *ExecView* aufgerufen werden. Siehe auch: *TGroup.ExecView*.

FindItem `function FindItem(Ch: Char): PMenuItem;`

Gibt einen Zeiger auf die Menüoption zurück, die über das Tastenkürzel *Ch* selektierbar ist, oder **nil**, wenn eine solche Option nicht existiert oder diese Option gerade deaktiviert ist. Bei der Angabe des Parameters *Ch* spielt Groß- oder Kleinschreibung keine Rolle. Das als Tastenkürzel definierte Zeichen wird auf dem Bildschirm hervorgehoben dargestellt. Siehe auch: *TMenuView.HotKey*.

GetItemRect
```
procedure GetItemRect(Item: PMenuItem; var R: TRect);
                      virtual;
```
Dies ist eine abstrakte Methode, die in den abgeleiteten Objekten *TMenuBar* und *TMenuBox* überschrieben wird. Sie gibt die Begrenzungen des von *Item* belegten Bereichs zurück; dadurch ist es möglich zu ermitteln, ob dieser Bereich mit der Maus angeklickt wurde. Siehe auch: *TMenuBar.GetItemRect*, *TMenuBox.GetItemRect*.

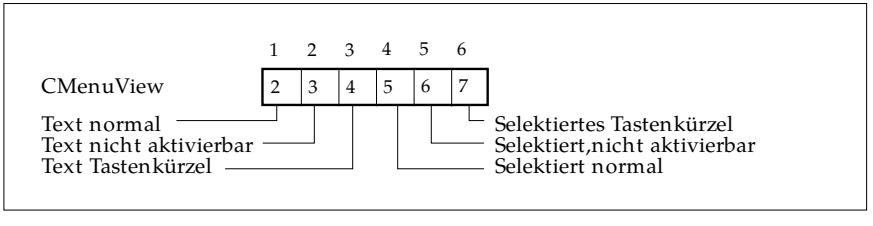
GetHelpCtx
```
function GetHelpCtx: Word; virtual;
```
Diese Methode gibt den im Feld *HelpCtx* gespeicherten Wert zurück. Ist dieser gleich *hcNoContext*, wird das entsprechende Feld des übergeordneten Menüs überprüft, darauf diejenigen eventuell weiterer übergeordneter *View*-Objekte. Enthalten auch diese *hcNoContext*, wird dieser Wert zurückgegeben. Siehe auch: *TView.HelpCtx*, *hcXXXX*-Konstanten.

GetPalette
```
function GetPalette: PPalette; virtual;
```
Gibt einen Zeiger auf die Farbpalette *CMenuBar* zurück.

HandleEvent
```
procedure HandleEvent(var Event: TEvent); virtual;
```
Wird immer aufgerufen, wenn Maus- oder Tastaturereignisse das Menü betreffen. Die Methode ermittelt zunächst, welche Option durch ein solches Ereignis angesprochen wurde, und generiert dann über *PutEvent* das passende Befehlsereignis. Siehe auch: *TView.HandleEvent*, *TView.PutEvent*.

HotKey
```
function HotKey(KeyCode: Word): PMenuItem;
```
Gibt einen Zeiger auf die Menüoption zurück, die über das in *KeyCode* angegebene Tastenkürzel selektierbar ist, oder **nil**, wenn eine solche Option nicht existiert oder diese Option gerade deaktiviert ist. Diese Tastenkürzel sind normalerweise Funktionstasten oder Kombinationen aus `[Alt][Buchstabe]` und werden über Aufrufe von *NewItem* und *NewSubMenu* innerhalb von *InitMenuBar* festgelegt. Über *HotKey* ermittelt die *HandleEvent*-Methode, welche Menüoption selektiert wurde. Siehe auch: *TApplication.InitMenuBar*, *TMenuView.FindItem*.

Store
```
procedure Store(var S: TStream);
```

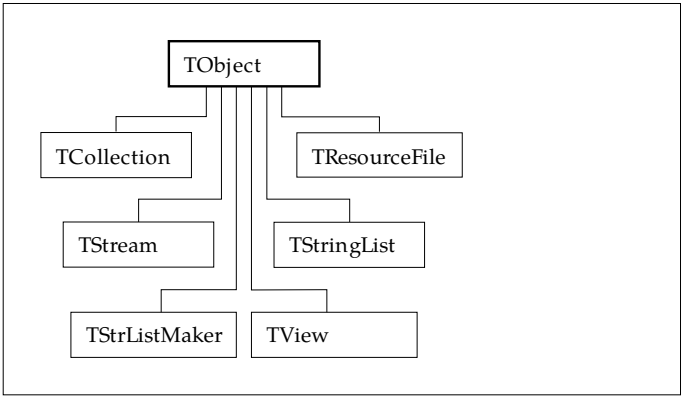


Schreibt ein *TMenuView*-Objekt in den angegebenen Stream. Siehe auch: *TMenuView.Load*.

Farbpalette Dieses Objekt greift über die Palette *CMenuView* auf die Einträge 2 bis 7 der aktuellen Farbpalette des Programms zu.

TObject

OBJECTS



TObject bildet den Ausgangspunkt in der Objekthierarchie von Turbo Vision. Als Basisobjekt hat *TObject* keinen Vorfahren, aber viele Nachkommen. Die Objekte *TPoint* und *TRect* ausgenommen, sind alle Standardobjekte, die Turbo Vision bereitstellt, von *TObject* abgeleitet. Jedes Objekt, das die Möglichkeiten eines *Stream* nutzen möchte, *muß* mittelbar oder unmittelbar von *TObject* abgeleitet sein.

Methoden

Init

```
constructor Init;
```

Reserviert für das Object Speicher auf dem Heap und löscht diesen Speicherbereich. Diese Methode wird von den Konstruktor-Methoden aller abgeleiteten Objekte aufgerufen. Da *TObject.Init* auch alle Felder der abgeleiteten Objekte löscht, müssen Sie *TObject.Init* aufrufen, *bevor* Sie in der Konstruktor-Methode des abgeleiteten Objekts irgendwelche Felder mit Werten belegen.

Free

```
procedure Free;
```

Die Methode *TObject.Free* gibt den von dem Objekt belegten Speicher wieder frei.

Done

```
destructor Done; virtual;
```

Gibt das Objekt frei; dieser Destruktor wird von *TObject.Free* aufgerufen.

TParamText DIALOGS

TParamText ist ein Nachkomme von *TStaticText* und dient dazu, über die Prozedur *FormatStr* Strings auszugeben, die Formatanweisungen enthalten.

Felder

ParamCount

```
ParamCount: Integer;
```

Gibt die Zahl der Parameter an, die in *ParamList^* gespeichert sind.

ParamList

```
ParamList: Pointer;
```

Dieser untypisierte Zeiger referenziert ein Array oder einen Record, die Zeiger oder LongInt-Werte aufnehmen und die notwendigen Parameter zur Formatierung des Strings liefern.

Methoden

Init

```
constructor Init(var Bounds: TRect; AText: String;
                 AParamCount: Integer);
```

Initialisiert über *TStaticText.Init* ein *TParamText*-Objekt in der in *Bounds* angegebenen Größe. Der String *AText* kann Formatanweisungen in der Form %[-][nnnn]X enthalten, die durch die zur Laufzeit des Programms übergebenen Parameter ersetzt werden. Die in *AParamCount* angegebene Zahl solcher Parameter wird im Feld *ParamCount* abgelegt. Die Formatanweisungen werden genauer im Eintrag zu *FormatStr* in Kapitel 15, »Außerhalb der Objekte«, behandelt. Siehe auch: *TStaticText.Init*, *FormatStr*.

Load

```
constructor Load(var S: TStream);
```

Liest über *TStaticText.Load* ein *TParamText*-Objekt aus dem angegebenen Stream. Auch der Wert *ParamCount* wird aus dem Stream geladen. Siehe auch: *TStaticText.Load*.

DataSize

```
function DataSize: Word; virtual;
```

Diese Methode gibt die Größe des Datenblocks zurück, den die Parameter erfordern, also *ParamCount* * *SizeOf*(*LongInt*).

GetText

```
procedure GetText(var S: String); virtual;
```

Formatiert über die Anweisung *FormatStr*(*S, Text^, ParamList^*) den übergebenen String *S*.

SetData

```
procedure SetData(var Rec); virtual;
```

Kopiert *DataSize* Bytes Daten aus *Rec* nach *ParamList*^. Siehe auch: *View.SetData*.

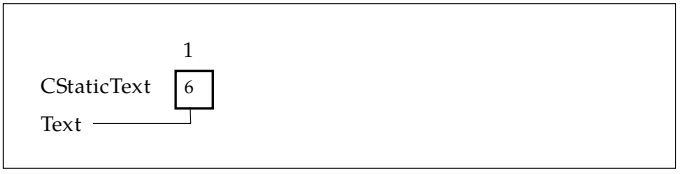
Store

```
procedure Store(var S: TStream);
```

Speichert über *TStaticText.Store* ein *TParamText*-Objekt, dann auch den Wert des Feldes *ParamCount* im Stream. Siehe auch: *TStaticText.Store*.

Farbpalette

Das Objekt *TParamText* benutzt die Palette *CStaticText*, über die der sechste Eintrag in der von *TDialog*-Objekten benutzten Standardpalette ermittelt wird.



TPoint

OBJECTS

Das Objekt *TPoint* repräsentiert einen Punkt auf dem Bildschirm.

Felder

X

```
X: Integer;
```

Speichert die X-Koordinate (Spalte) des Punktes auf dem Bildschirm.

Y

```
Y: Integer;
```

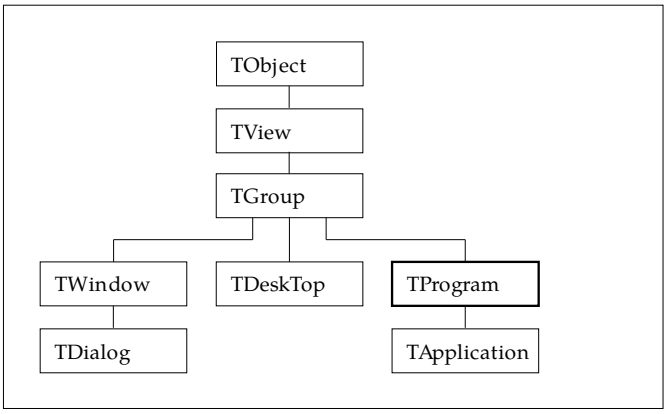
Speichert die Y-Koordinate (Zeile) des Punktes auf dem Bildschirm.

TProgram

APP

TProgram bildet das Fundament aller Turbo-Vision-Programme, da alle diese Programme von *TProgram* oder seinem Nachkommen *TApplication* abgeleitet werden müssen. *TApplication* unterscheidet sich von *TProgram* nur durch seine *Init*- und seine *Done*-Methode. Auf welchem dieser beiden Objekte Sie Ihr Programm aufbauen, hängt von Ihrer Anwendung ab. In der Regel wird aber ein Turbo-Vision-Programm von *TApplication* abgeleitet.

TProgram ist als Nachkomme von *TGroup* definiert, da diesem Objekt Instanzen von *TDeskTop*, *TStatusLine* und *TMenuBar* eingefügt werden müssen.



Methoden

Init `constructor Init;`

Setzt die globale Variable *Application* auf *@Self* und ruft die *Init*-Methode von *TGroup* auf, die ihrerseits die Gruppe *TProgram* initialisiert und ihr den ganzen Bildschirm als Bereich zuordnet. Im Feld *Status* werden die Bits *sfVisible*, *sfSelected*, *sfFocused*, *sfModal* und *sfExposed* gesetzt. Das Feld *Options* wird auf 0 gesetzt. Zur schnelleren Darstellung wird ein Bildschirmpuffer angelegt. Danach ruft *TProgram.Init* die Methoden *InitDeskTop*, *InitStatusLine* und *InitMenuBar* auf. Diese drei dadurch initialisierten Objekte werden anschließend in die Gruppe *TProgram* eingefügt. Siehe auch: *TGroup.Init*, *TProgram.InitDeskTop*, *TProgram.InitStatusLine*, *TProgram.InitMenuBar*.

Done `destructor Done; virtual;`

Gibt den von *TProgram* und den damit verbundenen Objekten belegten Speicher wieder frei und setzt die globale Variable *Application* auf **nil**. Jedes Programm muß diese Methode (oder die entsprechende Methode bei abgeleiteten Objekten) aufrufen, bevor der Programmlauf endet. Siehe auch: *TGroup.Done*.

EventError `procedure EventError(var Event: TEvent); virtual;`

Diese Methode wird immer dann aufgerufen, wenn ein Ereignis von *Subviews* oder Untergruppen nicht abgearbeitet wurde. Siehe auch: *TGroup.EventError*.

GetEvent `procedure GetEvent(var Event: TEvent); virtual;`

Die Methode *TView.GetEvent* beschränkt sich darauf, die *GetEvent*-Methode der Gruppe, zu der es gehört, aufzurufen. Da nun das *TProgram*-Objekt letzten Endes die Gruppe ist, zu der direkt oder indirekt alle *View*-Objekte gehören, wird

schließlich die Methode *TProgram.GetEvent* aufgerufen (es sei denn, die *GetEvent*-Methode irgendeines Objekts auf diesem Weg wäre überschrieben worden).

TProgram.GetEvents überprüft zunächst, ob ein von *TProgram.PutEvent* generiertes Ereignis anliegt und gibt gegebenenfalls dieses Ereignis zurück. Ist das nicht der Fall, wird *GetMouseEvent* und, wenn auch darüber nur ein Ereignis des Typs *evNothing* ermittelt wird, *GetKeyEvent* aufgerufen. Ergibt auch dies ein *evNothing*-Ereignis, ruft *GetEvent* die Methode *Idle* auf. Dadurch kann ein Programm, das auf Eingaben des Benutzers wartet, im Hintergrund andere Aufgaben erledigen. Ist endlich ein der Weitergabe würdiges Ereignis ermittelt, überprüft *GetEvent*, ob dies ein Ereignis des Typs *evKeyDown* oder *evMouseDown* ist. Sollte das der Fall sein, wird das Ereignis zunächst dem *TStatusLine*-Objekt übergeben. Dieses Objekt ordnet gegebenenfalls dieses Ereignis einem der Befehle zu, die über die Statuszeile erreichbar sind, und gibt ein *evCommand*-Ereignis zurück.

Siehe auch: *TProgram.PutEvent*, *GetMouseEvent*, *GetKeyEvent*.

GetPalette

```
function GetPalette: PPalette; virtual;
```

Gibt einen Zeiger auf die aktuell verwendete Farbpalette zurück. *TProgram* unterstützt drei Farbpaletten: *apColor*, *apBlackWhite* und *apMonochrome*. Die Methode *TProgram.InitScreen* legt fest, welche Palette verwendet wird, und speichert deren Nummer in der globalen Variablen *AppPalette*. Siehe auch: *TProgramm.InitScreen*, *AppPalette*, *apXXXX*-Konstanten.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Dies ist die zentrale Methode zur Abarbeitung von Ereignissen; zuständig ist sie für Tastenkürzel, die im gesamten Programm verfügbar sind. Siehe auch: *TGroup.HandleEvent*.

Idle

```
procedure Idle; virtual;
```

Diese abstrakte Methode wird immer dann aufgerufen, wenn keine Ereignisse zur Bearbeitung anstehen. Mittels dieser Methode können Sie zum Beispiel die Statuszeile neu aufbauen lassen, um etwa die aktuelle Uhrzeit oder den noch verfügbaren Speicher in ihr anzeigen zu lassen. Um dies zu erreichen, müssen Sie natürlich diese Methode überschreiben.

InitDeskTop

```
procedure InitDeskTop; virtual;
```

Wird von *TProgram.Init* zur Initialisierung eines *TDeskTop*-Objekts aufgerufen und speichert einen Zeiger auf dieses

Objekt in der globalen Variablen *Desktop*. Siehe auch: *TProgram.Init, TDeskTop*.

InitMenuBar `procedure InitMenuBar; virtual;`

Wird von *TProgram.Init* zur Initialisierung der Menüleiste aufgerufen und speichert einen Zeiger auf dieses Objekt in der globalen Variablen *MenuBar*. Siehe auch: *TProgram.Init, TMenuBar*.

InitScreen `procedure InitScreen; virtual;`

Wird von *TProgram.Init* und *TProgram.SetScreenMode* immer dann aufgerufen, wenn der Bildschirmmodus geändert wird. Diese Methode paßt die Werte aller vom Bildschirmmodus abhängigen Variablen neu an. Siehe auch: *TProgram.Init, TProgram.SetScreenMode*.

InitStatusLine `procedure InitStatusLine; virtual;`

Wird von *TProgram.Init* zur Initialisierung der Statuszeile aufgerufen und speichert einen Zeiger auf dieses Objekt in der globalen Variablen *StatusLine*. Siehe auch: *TProgram.Init, TStatusLine*.

OutOfMemory `procedure OutOfMemory; virtual;`

TProgram.ValidView ruft diese Methode auf, wenn die Variable *LowMemory* auf True gesetzt ist. Die Methode *OutOfMemory* sollte den Benutzer darüber informieren, daß zur Ausführung der gewählten Operation nicht mehr genügend Speicher zur Verfügung steht. Dazu kann zum Beispiel die Routine *MessageBox* aus der Unit *StdDlg* verwendet werden:

```
procedure TMyApp.OutOfMemory;
begin
  MessageBox('Nicht gen¡gend Speicher, um diese Operation
              auszuf¡hren.', nil, mfError + mfOkButton);
end;
```

Siehe auch: *TProgram.ValidView, LowMemory*.

PutEvent `procedure PutEvent(var Event: TEvent); virtual;`

Reiht das Ereignis *Event* in die Warteschlange ein. Diese Methode wird dazu benutzt, ein gerade unerwünschtes Ereignis zur späteren Bearbeitung zurückzustellen, oder auch dazu, zur Kommunikation zwischen *View*-Objekten ein Befehlsereignis zu generieren. Siehe auch: *TView.PutEvent, TProgram.GetEvent*.

Run `procedure Run; virtual;`

Diese Methode startet über einen Aufruf von *Execute* das

Programm und sorgt gleichzeitig dafür, daß *TProgramm* modal wird. Siehe auch: *TGroup.Execute*.

SetScreenMode

```
procedure SetScreenMode(Mode: Word);
```

Ändert den Bildschirmmodus; über diese Methode können Sie zum Beispiel vom 25-Zeilen- in den 43/50-Zeilen-Modus schalten. Siehe auch: *TProgram.InitScreen*, *SetVideoMode*, *smXXXX*-Konstanten.

ValidView

```
function ValidView(P: PView): PView;
```

Überprüft die Gültigkeit eines gerade instantiierten *View*-Objekts und gibt *P* zurück, wenn das Objekt gültig ist, andernfalls **nil**. Dies kann drei Ursachen haben: Einmal kann der Parameter P selbst **nil** sein. Zweitens ist es möglich, daß LowMemory auf True gesetzt ist; dann gibt *ValidView* den von diesem Objekt belegten Speicher frei. Drittens kann der Aufruf von *P^.Valid(cmValid)* False zurückgeben, und auch in diesem Fall gibt ValidView den von diesem Objekt belegten Speicher frei.

ValidView wird oft benutzt, um die Gültigkeit eines Objekts zu überprüfen, bevor es in eine Gruppe eingefügt wird. Die folgende Anweisung instantiiert ein Fensterobjekt, überprüft seine Gültigkeit und fügt es in die *DeskTop*-Gruppe ein:

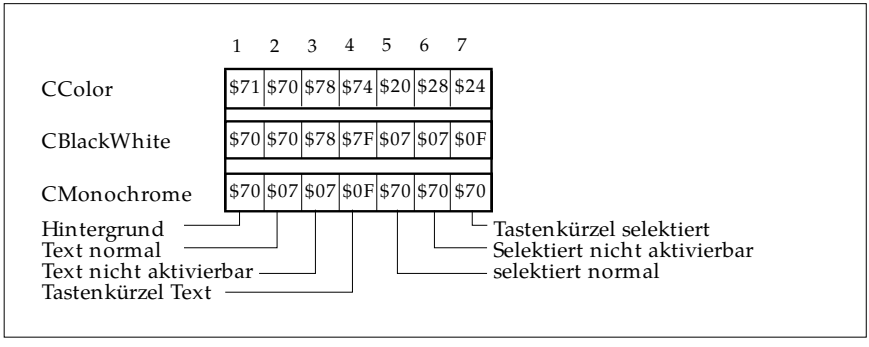
```
DeskTop^.Insert(ValidView(New(TMyWindow, Init(...))));
```

Siehe auch: *LowMemory*, *TProgram.OutOfMemory*.

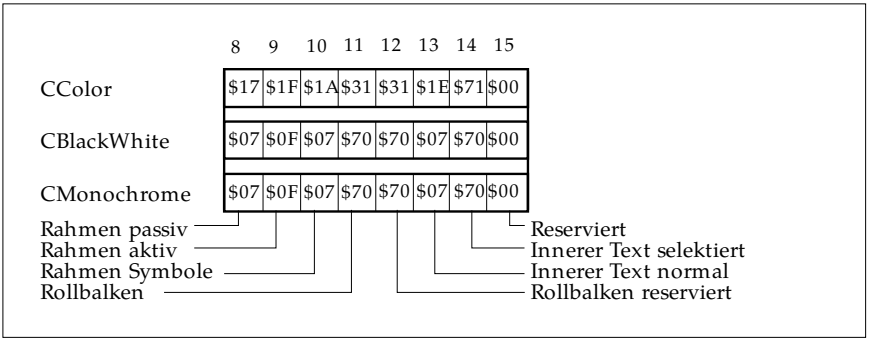
Farbpalette

Die Farbpaletten, die *TProgram* definiert, enthalten die tatsächlich in einem Turbo-Vision-Programm verwendeten Bildschirmattribute, auf die sich andere *View*-Objekte über die Indices ihrer Farbpaletten beziehen.

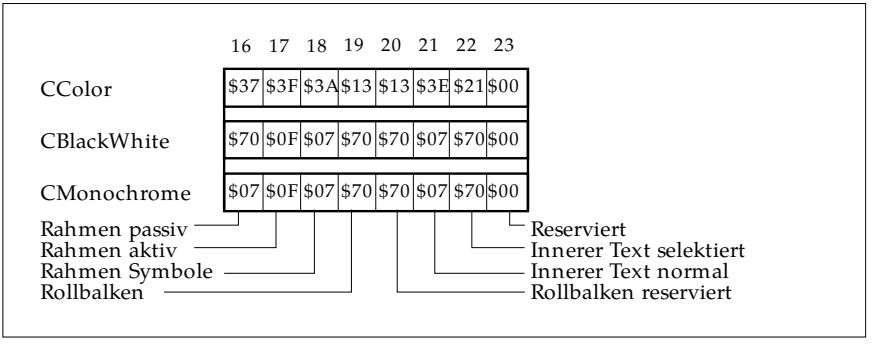
Der erste Eintrag wird von *TBackGround*, die Einträge 2 bis 7 von Menüs und von der Statuszeile verwendet:



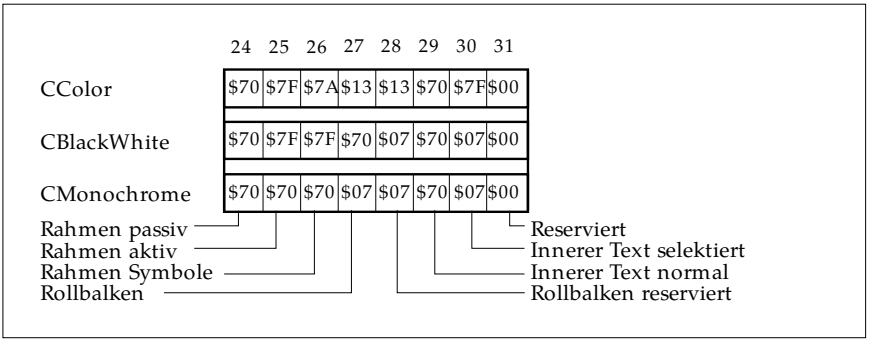
Die Einträge 8 bis 15 verwenden *TWindow*-Objekte für ihre Farbpalette *wpBlueWindow*:



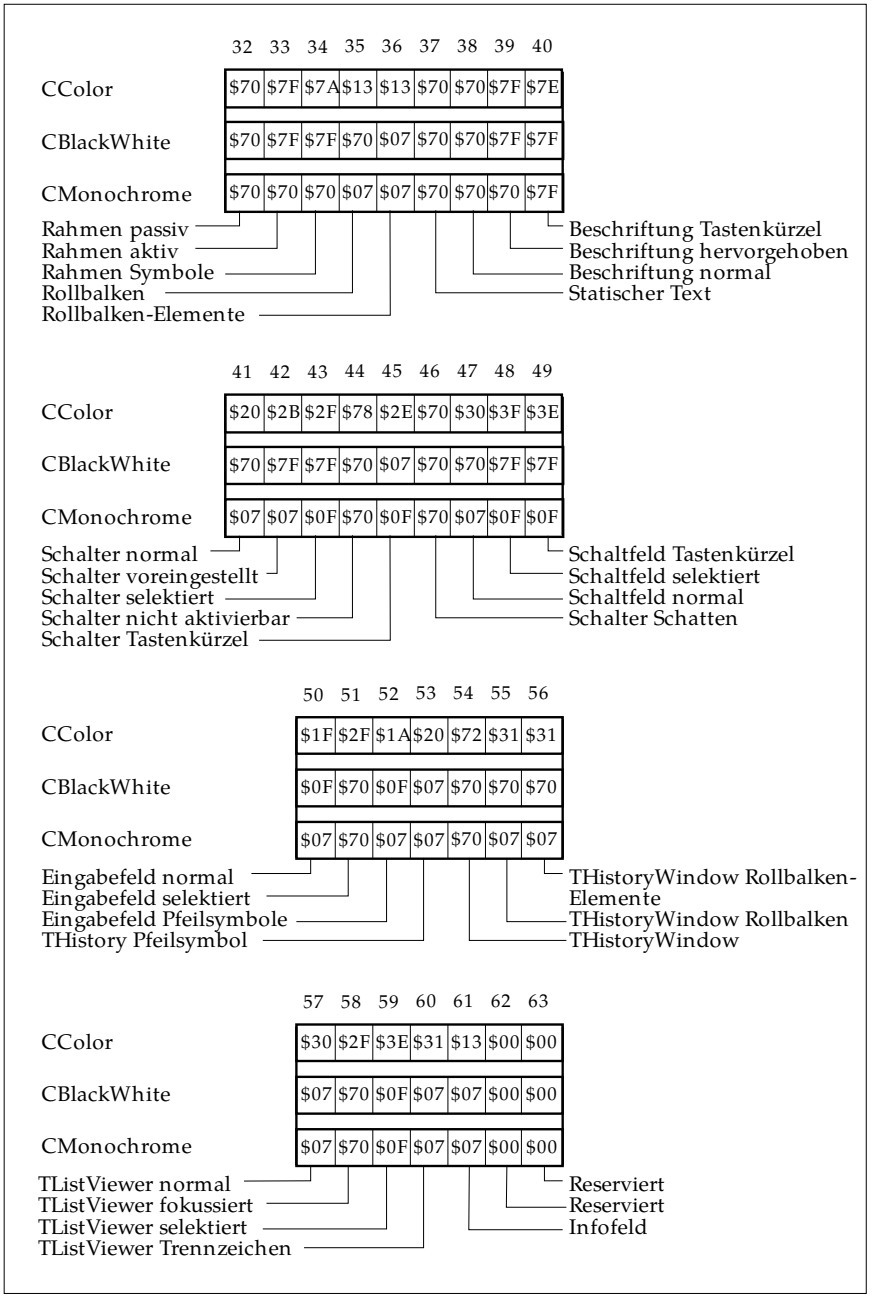
Die Einträge 16 bis 23 verwenden *TWindow*-Objekte für ihre Farbpalette *wpCyanWindow*:



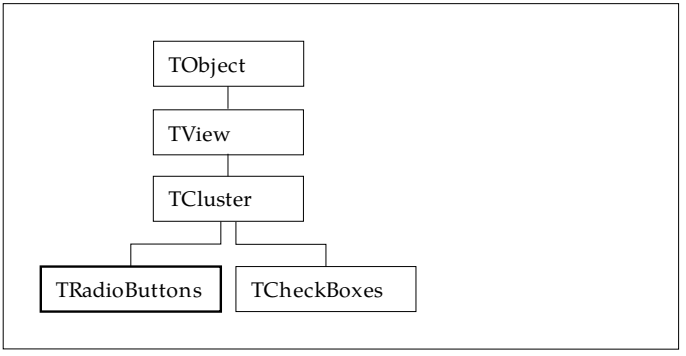
Die Einträge 24 bis 31 verwenden *TWindow*-Objekte für ihre Farbpalette *wpGrayWindow*:



Die Einträge 32 bis 63 werden von *TDialog*-Objekten verwendet:



TRadioButtons DIALOGS



TRadioButtons stellt ein Auswahlfeld zur Verfügung, innerhalb dessen immer nur ein Schalter aktiv sein kann. Wird ein Schalter »gedrückt«, wird automatisch der vorher aktivierte Schalter gelöst. Seine Funktionalität bezieht dieses Objekt fast gänzlich aus *TCluster*, die Methoden *Init*, *Load* und *Done* eingeschlossen. Objekte des Typs *TRadioButtons* sind meist mit *TLabel*-Objekten verbunden, wodurch die Anwahl eines Schalters über die Selektion der mit ihm verbundenen Beschriftung möglich wird.

Felder

Über die von *TCluster* geerbten Felder hinaus definiert dieses Objekt keine weiteren Felder, interpretiert allerdings das Feld *Values* auf spezielle Weise: Es gibt die Nummer des gewählten Schalters an, wobei diese Schalter mit 0 beginnend gezählt werden.

Methoden

Draw

```
procedure Draw; virtual;
```

Stellt die Schalter als einen von runden Klammern ( ) umrahmten Bereich dar. Wollen Sie eine andere Darstellung, müssen Sie diese Methode überschreiben.

Mark

```
function Mark(Item: Integer): Boolean; virtual;
```

Gibt True zurück, wenn der Wert im Feld *Value* gleich *Item* ist. Diese Methode geht von der einfachsten Zuordnung zwischen Schalter und *Item* aus, nämlich dem der Gleichheit. Brauchen Sie eine komplexere Zuordnung, müssen Sie diese Methode überschreiben. Siehe auch: *TCluster.Value*, *TCluster.Mark*.

MovedTo

```
procedure MovedTo(Item: Integer); virtual;
```

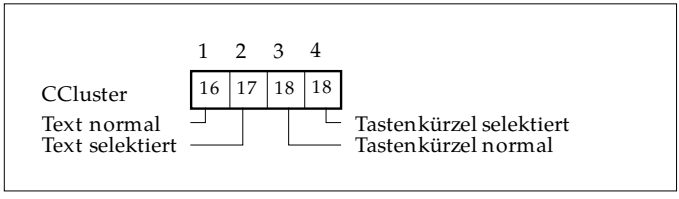
Setzt das Feld *Value* auf den Wert *Item*. Diese Methode geht von der einfachsten Zuordnung zwischen Schalter und *Item*

aus, nämlich dem der Gleichheit. Brauchen Sie eine komplexere Zuordnung, müssen Sie diese Methode überschreiben. Siehe auch: *TCluster.MovedTo*, *TRadioButtons.Mark*.

Press
```
procedure Press(Item: Integer); virtual;
```
Setzt das Feld *Value* auf den Wert *Item* und wird immer dann aufgerufen, wenn der Schalter Nummer *Item* »gedrückt« wird. Diese Methode geht von der einfachsten Zuordnung zwischen Schalter und *Item* aus, nämlich dem der Gleichheit. Siehe auch: *TCluster.Press*.

SetData
```
procedure SetData(var Rec); virtual;
```
Kopiert den Wert von *Value* aus *Rec*. Siehe auch: *TCluster.SetData*, *TCluster.GetData*, *TCluster.DataSize*.

Farbpalette Dieses Objekt greift über die Palette *CCluster* auf die Einträge 16 bis 18 der Palette *CDialog* zu.



TRect OBJECTS

Felder

A
```
A: TPoint;
```
Speichert die Koordinaten der oberen linken Ecke eines rechteckigen Bereiches auf dem Bildschirm.

B
```
B: TPoint;
```
Speichert die Koordinaten der unteren rechten Ecke eines rechteckigen Bereiches auf dem Bildschirm.

Methoden

Assign
```
procedure Assign(XA, YA, XB, YB: Integer);
```
Speichert die übergebenen Werte *XA* und *YA* im Feld *A*, die Werte *XB* und *YB* im Feld *B*.

Copy
```
procedure Copy(R: TRect);
```
Setzt die Felder *A* und *B* auf die in *R* übergebenen Werte.

Move
```
procedure Move(ADX, ADY: Integer);
```

Verändert die Position des Rechtecks, indem *ADX* zu *A.X* und *B.X*, und *ADY* zu *A.Y* und *B.Y* addiert wird.

Grow

```
procedure Grow(ADX, ADY: Integer);
```

Diese Methode ändert die Größe des Bereichs, indem sie *ADX* von *A.X* subtrahiert und zu *B.X* addiert, sowie *ADY* von *A.Y* subtrahiert und zu *B.Y* addiert.

Intersect

```
procedure Intersect(R: TRect);
```

Diese Methode ermittelt den Bereich, der sowohl in dem durch die Felder *A* und *B* festgelegten Rechteck als auch in dem durch *R.A* und *R.B* festgelegten Rechteck enthalten ist, und speichert die Koordinaten dieses Bereichs in den Feldern *A* und *B*.

Union

```
procedure Union(R: TRect);
```

Diese Methode ermittelt den Bereich, der sowohl das durch die Felder *A* und *B* festgelegte Rechteck als auch das durch *R.A* und *R.B* festgelegte Rechteck einschließt, und speichert die Koordinaten dieses Bereichs in den Feldern *A* und *B*.

Contains

```
function Contains(P: TPoint): Boolean;
```

Gibt True zurück, wenn sich der Punkt *P* innerhalb des durch die Felder *A* und *B* definierten Rechtecks befindet.

Equals

```
function Equals(R: TRect): Boolean;
```

Gibt True zurück, wenn die Werte in *R* den Werten der Felder *A* und *B* entsprechen.

Empty

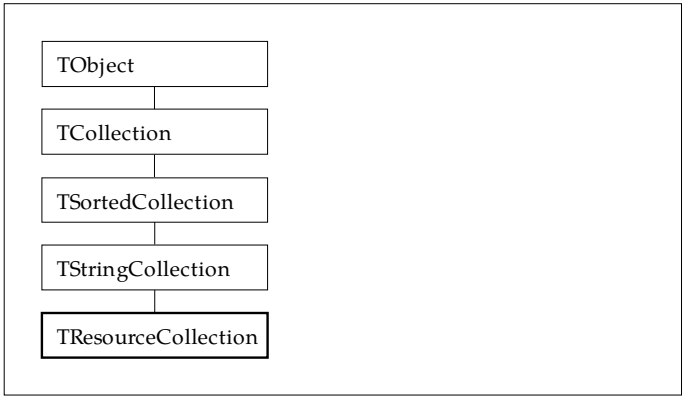
```
function Empty: Boolean;
```

Gibt True zurück, wenn die Werte in *A* denen in *B* entsprechen, der dadurch definierte Bereich also keine Ausdehnung besitzt.

TResourceCollection

OBJECTS

TResourceCollection ist ein Nachkomme von *TStringCollection* und stellt in Zusammenarbeit mit *TResourceFile* eine Kollektion bereit, die Ressourcen speichert. Eine Ressourcen-Datei ist nichts anderes als ein *Stream*, indiziert über einen String. Jedes Element einer Ressourcen-Kollektion verfügt über das Feld *Pos* vom Typ Integer und das Schlüsselfeld *Key* vom Typ String. Einzig um dieses Schlüsselfeld richtig behandeln zu können, überschreiben die Methoden von *TResourceCollection* die entsprechenden Methoden ihrer Vorfahren. *TResourceCollection* wird von dem Objekt *TResourceFile* intern zur Verwaltung des Indexes einer Ressourcen-Datei verwendet.



TResourceFile

OBJECTS

TResourceFile stellt einen Stream zur Verfügung, dessen Einträge über einen Schlüssel des Typs String indiziert sind. Sie können mittels der Methode *TResourceFile.Get*, der Sie den Schlüssel-String übergeben, auf einen Eintrag zugreifen. Über *TResourceFile.Put* schreiben Sie einen Eintrag in die Ressourcen-Datei.

Felder

Stream

```
Stream: PStream;
```

Zeiger auf den Stream, der mit der Ressourcen-Datei verbunden ist.

Modified

```
Modified: Boolean;
```

Wird auf True gesetzt, wenn die Ressourcen-Datei geändert wurde.

Methoden

Init

```
constructor Init(AStream: PStream);
```

Erstellt eine Ressourcen-Datei mit dem angegebenen Stream,

speichert *AStream* im Feld *Stream* und setzt das Feld *Modified* auf False. Der Stream muß bereits initialisiert sein, zum Beispiel über die Anweisung:

```
ResFile.Init(New(TBufStream, Init('MYAPP.RES',
             stOpenRead, 1024)));
```

Während der Initialisierung sucht *Init* an der aktuellen Position im Stream nach dem Dateikopf der Ressourcen-Datei. Das Format dieses Dateikopfes ist folgendermaßen festgelegt:

```
type
  TResFileHeader = record
    Signature: array[1..4] of Char;
    ResFileSize: LongInt;
    IndexOffset: LongInt;
  end;
```

Signature enthält die Kennung 'FBPR', *ResFileSize* die Größe der Ressourcen-Datei minus 8 Bytes, da *Signature* und *ResFileSize* von der Größe der Gesamtdatei abgezogen werden. *IndexOffset* enthält den Offset auf den Beginn des in der Datei gespeicherten Index, berechnet vom Beginn des Dateikopfes.

Findet *Init* an der aktuellen Position im Stream diesen Dateikopf nicht, geht die Methode davon aus, daß eine neue Ressourcen-Datei angelegt wurde, und initialisiert einen leeren Index.

Findet *Init* aber an der aktuellen Position im Stream die Signatur einer .EXE-Datei, überliest die Methode alle ausführbaren Daten und sucht an der neuen Position im Stream nach dem Dateikopf der Ressourcen-Datei. Die Methode geht genauso vor, wenn sie die Signatur einer Overlay-Datei findet, die an eine .EXE-Datei angehängt wurde. Und genauso verhält sich *OvrInit*; auch diese Standardprozedur überspringt eine angehängte Ressourcen-Datei. Das bedeutet, daß Sie an das Ende der .EXE-Datei Ihres Programms sowohl eine Ressourcen- als auch eine Overlay-Datei anhängen können, und das auch in beliebiger Reihenfolge.

Siehe auch: *TResourceFile.Done*.

Done

```
destructor Done; virtual;
```

Ruft die Methode *Flush* auf, um alle Änderungen zu sichern, und gibt das Objekt frei. Siehe auch: *TResourceFile.Init*, *TResourceFile.Flush*.

Count

```
function Count: Integer;
```

Gibt die Zahl der Einträge in der Ressourcen-Datei zurück. Siehe auch: *TResourceFile.KeyOf*.

Delete

```
procedure Delete(Key: String);
```

Löscht den über *Key* referenzierten Eintrag in der Ressourcen-Datei. Der von diesem Eintrag belegte Speicherplatz wird nicht freigegeben. Sie können aber über *SwitchTo* eine bereinigte Kopie der Ressourcen-Datei erstellen. Siehe auch: *TResourceFile.SwitchTo*.

Flush

```
procedure Flush;
```

Wenn die Ressourcen-Datei geändert wurde, was über das Feld *Modified* ermittelt wird, schreibt *Flush* den geänderten Index an das Ende des Stream und ändert auch den Dateikopf am Anfang des Stream. Darauf wird *Modified* wieder auf False gesetzt. Siehe auch: *Modified*, *TResourceFile.Done*.

Get

```
function Get(Key: String): PObject;
```

Ermittelt über *TSortedCollection.Search* einen Zeiger auf das durch *Key* indizierte, in der Ressourcen-Datei gespeicherte Objekt. Wird ein solcher Eintrag nicht gefunden, gibt die Methode **nil** zurück. Zum Beispiel:

```
DeskTop^.Insert(ValidView(ResFile.Get
                ('EditorWindow')));
```

Siehe auch: *TResourceFile.KeyAt*, *TResourceFile.Put*.

KeyAt

```
function KeyAt(I: Integer): String;
```

Gibt den Schlüssel-String für den Eintrag mit der Nummer *I* in der Ressourcen-Datei zurück. Die Indexnummer des ersten Eintrags ist Null, die des letzten *TResourceFile.Count* - 1. Siehe auch: *TResourceFile.Count*.

Put

```
procedure Put(Item: PObject; Key: String);
```

Schreibt das übergebene Objekt in die Ressourcen-Datei und indiziert es mittels des Schlüssels *Key*. Siehe auch: *TResourceFile.Get*.

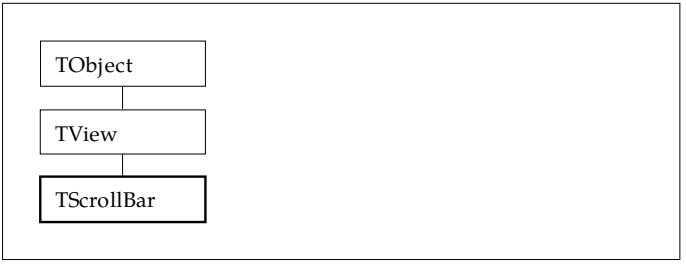
SwitchTo

```
function SwitchTo(AStream: PStream; Pack: Boolean):
                  PStream;
```

Kopiert die Ressourcen-Datei in den Stream *AStream* und gibt einen Zeiger auf den Original-Stream zurück.

Ist der Parameter *Pack* True, werden leere oder nicht genutzte Einträge nicht in den neuen Stream kopiert. Das ist der einzige Weg, eine Ressourcen-Datei zu komprimieren. Wenn Sie *Pack* auf False setzen, wird schneller kopiert, jedoch keine Komprimierung durchgeführt.

TScrollBar



Felder

Value

```
Value: Integer;
```

Dieses Feld speichert die aktuelle Position des Zeichens auf dem Rollbalken, das als Indikator für die relative Position der dargestellten Daten innerhalb des gesamten darstellbaren Datenbereichs dient. Viele Maus- oder Tastaturereignisse verändern mittelbar oder unmittelbar den Wert dieses Feldes, zum Beispiel wenn mit der Maus der Rollbalken angeklickt oder die Größe des Fensters geändert wird. Umgekehrt kann auch eine Änderung von *Value* dazu führen, daß weitere Ereignisse generiert werden. Der Konstruktor *Init* setzt dieses Feld auf 0. Siehe auch: *TScrollBar.SetValue*, *TScrollBar.SetParams*, *TScrollBar.ScrollDraw*, *TScroller.HandleEvent*, *TScrollBar.Init*.

Min

```
Min: Integer;
```

Der Minimalwert, den *Value* nicht unterschreiten darf. Der Konstruktor *Init* setzt dieses Feld auf 0. Siehe auch: *TScrollBar.SetRange*, *TScrollBar.SetParams*.

Max

```
Max: Integer;
```

Der Maximalwert, den *Value* nicht überschreiten darf. Der Konstruktor *Init* setzt dieses Feld auf 0. Siehe auch: *TScrollBar.SetRange*, *TScrollBar.SetParams*.

PgStep

```
PgStep: Integer;
```

Dieser Wert wird zu *Value* addiert oder davon subtrahiert, wenn der Rollbalken mit der Maus angeklickt oder ein dazu gleichwertiger Tastendruck (`[Ctrl][Left]`, `[Ctrl][Right]` , `[PgUp]` , `[PgDn]`) registriert wird. Der Konstruktor *Init* setzt dieses Feld auf 1. *PgStep* kann über die Methoden *TScrollBar.SetStep*, *TScrollBar.SetParams* oder *TScrollBar.SetLimit* geändert werden. Siehe auch: *TScrollBar.SetStep*, *TScrollBar.SetParams*, *TScrollBar.SetLimits*, *TScrollBar.ScrollStep*.

ArStep
```
ArStep: Integer;
```
Dieser Wert wird zu *Value* addiert oder davon subtrahiert, wenn ein Pfeilsymbol mit der Maus angeklickt oder ein dazu gleichwertiger Tastendruck registriert wird. Der Konstruktor *Init* setzt dieses Feld auf 1. Siehe auch: *TScrollBar.SetStep*, *TScrollBar.SetParams*, *TScrollBar.ScrollStep*.

Methoden

Init
```
constructor Init(var Bounds: TRect);
```
Initialisiert einen Rollbalken von der in *Bounds* angegebenen Größe. Den Feldern *Value*, *Min*, *Max* wird 0, den Feldern *PgStep* und *ArStep* wird 1 zugewiesen. Ergibt *Bounds* für *X* den Wert 1, dann erhalten Sie einen vertikalen Rollbalken, andernfalls einen horizontalen. Im Feld *GrowMode* eines vertikalen Rollbalkens sind die Bits *gfGrowLoX*, *gfGrowHiX* und *gfGrowHiY*, in dem eines horizontalen Rollbalkens die Bits *gfGrowLoY*, *gfGrowHiX* und *gfGrowHiY* gesetzt.

Load
```
constructor Load(var S: TStream);
```
Lädt ein *TScrollBar*-Objekt aus dem angegebenen Stream. Siehe auch: *TScrollBar.Store*.

Draw
```
procedure Draw; virtual;
```
Stellt den Rollbalken unter Berücksichtigung der Werte in *Value* und *Value* auf dem Bildschirm dar. Siehe auch: *Value*, *TScrollBar.ScrollDraw*.

GetPalette
```
function GetPalette: PPalette; virtual;
```
Diese pseudo-abstrakte Methode gibt einen Zeiger auf die Farbpalette *CScrollBar* zurück.

HandleEvent
```
procedure HandleEvent(var Event: TEvent); virtual;
```
Ruft *TView.HandleEvent* und überprüft anschließend das Feld *Event.What*. Ergibt sich, daß ein Mausereignis vorliegt, wird es über die global verfügbare Funktion *Message* dem Objekt übergeben, zu dem der Rollbalken gehört. Dieses Objekt muß die anderen Objekte benachrichtigen, wenn sich Veränderungen im Status des Rollbalkens ergeben. Diese Objekte wiederum müssen alle Aktionen veranlassen, also zum Beispiel einen anderen Teil des Textes ausgeben. *HandleEvent* stellt ebenfalls fest, welcher Teil des Rollbalkens mit der Maus angeklickt wurde oder ob ein dazu gleichwertiger Tastenbefehl vorliegt. Darauf wird das Feld *Value* unter Berücksichtigung der Werte in *ArStep* und *PgStep* richtiggestellt und die Positionsmarkierung auf dem Rollbalken entsprechend gesetzt. Siehe auch: *TView.HandleEvent*.

ScrollDraw

```
procedure ScrollDraw; virtual;
```

Wann immer der Wert in *Value* sich ändert, wird diese pseudo-abstrakte Methode aufgerufen, die den Befehl *cmScrollBarChanged* an das Objekt sendet, zu dem der Rollbalken gehört:

```
Message(Owner, evBroadCast, cmScrollBarChanged, @Self);
```

Siehe auch: *Value*, *Message*.

ScrollStep

```
function ScrollStep(Part: Integer): Integer; virtual;
```

Diese Methode gibt einen positiven oder negativen Wert zurück, der einerseits von dem Argument *Part*, andererseits von den Werten in *ArStep* und *PgStep* abhängt. Für *Part* können folgende Konstanten verwendet werden:

```
sbLeftArrow  = 0;
sbRightArrow = 1;
sbPageLeft   = 2;
sbPageRight  = 3;
sbUpArrow    = 4;
sbDownArrow  = 5;
sbPageUp     = 6;
sbPageDown   = 7;
sbIndicator  = 8;
```

Siehe auch: *TScrollBar.SetStep*, *TScrollBar.SetParams*, *sbXXXX*-Konstanten.

SetParams

```
procedure SetParams(AValue, AMin, AMax, APgStep,
                    AArStep: Integer);
```

Diese Methode setzt die Felder *Value*, *Min*, *Max*, *PgStep* und *ArStep* auf die übergebenen Werte. Sind diese widersprüchlich, werden sie richtiggestellt. Zum Beispiel kann *Min* nicht größer als *Max* werden; wenn also *AMax* < *AMin*, wird *Max* auf *AMin* gesetzt. *Value* muß innerhalb des von *Min* und *Max* definierten Bereichs liegen; ist *AValue* < *AMin* wird *Value* *AMin*, ist *AValue* > *AMax* wird *Value* *AMax* zugewiesen. Der Rollbalken wird über *DrawView* erneut dargestellt. Wenn sich *Value* änderte, wird *ScrollDraw* aufgerufen. Dies löst möglicherweise bei dem Objekt, zu dem der Rollbalken gehört, weitere Ausgabeaktivitäten aus. Siehe auch: *TView.DrawView*, *TScrollBar.ScrollDraw*, *TScrollBar.SetRange*, *TScrollBar.SetValue*.

SetRange

```
procedure SetRange(AMin, AMax: Integer);
```

Diese Methode setzt die Felder *Min* und *Max* auf die Werte *AMin* und *AMax*, und ruft dann die Methode *SetParams* auf, so daß alle durch diese Änderungen notwendigen Ausgabe-

aktionen durchgeführt werden. Siehe auch: *TScrollBar.SetParams*, *TScrollBar.ScrollStep*.

SetStep

```
procedure SetStep(APgStep, AArStep: Integer);
```

Diese Methode setzt die Felder *PgStep* und *ArStep* auf die Werte *APgStep* und *AArStep*, und ruft dann die Methode SetParams auf, so daß alle durch diese Änderungen notwendigen Ausgabeaktionen durchgeführt werden. Siehe auch: *TScrollBar.SetParams*, *TScrollBar.ScrollStep*.

SetValue

```
procedure SetValue(AValue: Integer);
```

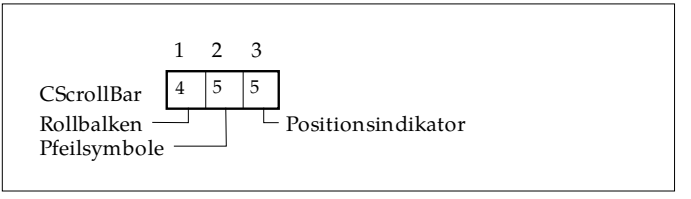
Diese Methode setzt das Feld *Value* auf den Wert *AValue* und ruft dann die Methode *SetParams* auf, so daß alle durch diese Änderung notwendigen Ausgabeaktionen durchgeführt werden. Siehe auch: *TScrollBar.SetParams*, *TScrollBar.ScrollDraw*, *TScroller.ScrollTo*, *TView.DrawView*.

Store

```
procedure Store(var S: TStream);
```

Speichert ein *TScrollBar*-Objekt im angegebenen Stream. Siehe auch: *TScrollBar.Load*.

Farbpalette Dieses Objekt greift über die Palette *CScrollBar* auf die Einträge 4 und 5 der aktuellen Farbpalette des Programms zu.



TScroller VIEWS

Felder

HScrollBar

```
HScrollBar: PScrollBar;
```

Zeigt auf den mit diesem Objekt verbundenen horizontalen Rollbalken oder enthält den Wert **nil**, wenn kein Rollbalken definiert ist.

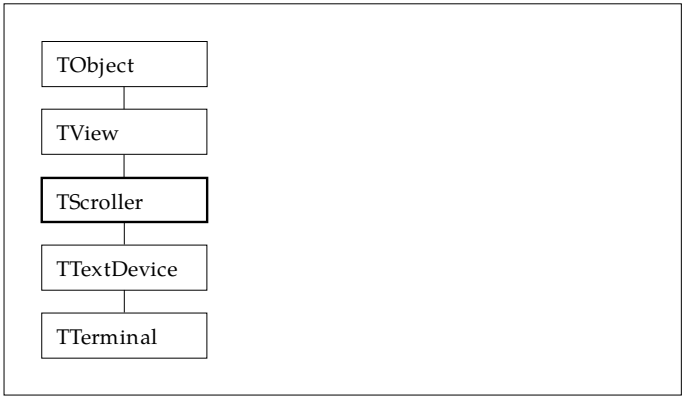
VScrollBar

```
VScrollBar: PScrollBar;
```

Zeigt auf den mit diesem Objekt verbundenen vertikalen Rollbalken oder enthält den Wert **nil**, wenn kein Rollbalken definiert ist.

Delta

```
Delta: TPoint;
```



Delta speichert in *X* die vertikale und in *Y* die horizontale relative Position des Darstellungsbereichs bezogen auf den gesamten darzustellenden Datenbereich. Die Bewegung des Fensters über diesen Datenbereich wird durch Änderung eines oder beider *Delta*-Werte veranlaßt. Ein solche Änderung kann zum Beispiel als Reaktion auf einen Mausklick innerhalb eines Rollbalkens erfolgen, dessen Feld *Value* sich infolge dieses Mausereignisses ändert. Umgekehrt führt eine Änderung von *Delta* auch zu einer Änderung des *Value*-Feldes des betroffenen Rollbalkens, was dazu führt, daß dessen Positionsmarkierung neu positioniert wird. Siehe auch: *TScroller.ScrollDraw*, *TScroller.ScrollTo*.

Limit

```
Limit: TPoint;
```

Limit.X und *Limit.Y* speichern die Werte, die *Delta.X* und *Delta.Y* nicht überschreiten dürfen. Siehe auch: *TScroller.Delta*.

Methoden

Init

```
constructor Init(var Bounds: TRect; AHScrollBar,
                 AVScrollBar: PScrollBar);
```

Initialisiert über *TView.Init*(*Bounds*) ein *TScroller*-Objekt in der in *Bounds* angegebenen Größe versehen mit den übergebenen Rollbalken, sofern die entsprechenden Argumente nicht **nil s**ind. In Options wird das Bit *ofSelectable* und das Feld *EventMask* wird auf *evBroadcast* gesetzt. Siehe auch: *TView.Init*, *TView.Options*, *TView.EventMask*.

Load

```
constructor Load(var S: TStream);
```

Lädt ein *TSroller*-Objekt und – sofern vorhanden – die damit verbundenen Rollbalken aus dem angegebenen Stream. Siehe auch: *TScroller.Store*.

ChangeBounds `procedure ChangeBounds(var Bounds: TRect); virtual;`

Ändert über einen Aufruf von *SetBounds*(*Bounds*) die Größe des Objekts; wenn notwendig, wird es, zusammen mit den verbundenen Rollbalken, erneut dargestellt. Siehe auch: *TView.SetBounds*, *TScroller.SetLimit*.

GetPalette `function GetPalette: PPalette; virtual;`

Diese pseudo-abstrakte Methode gibt einen Zeiger auf die Farbpalette *CScroller* zurück.

HandleEvent `procedure HandleEvent(var Event: TEvent); virtual;`

Diese Methode ist zuständig für alle dieses Objekt betreffenden Ereignisse. Enthält das Feld *Event.What* den Wert *evBroadCast* und das Feld *Event.Command* den Wert *brScrollBarChanged*, wird *Event.InfoPtr* überprüft. Zeigt dieses auf *HScrollBar* oder *VScrollBar*, wird das *TScroller*-Objekt, zusammen mit den verbundenen Rollbalken, mittels *TScroller.ScrollDraw* erneut dargestellt. Siehe auch: *TView.HandleEvent*, *TScroller.ScrollDraw*.

ScrollDraw `procedure ScrollDraw; virtual;`

Stellt das Objekt immer dann neu dar, wenn das Feld *Value* der verbundenen Rollbalken sich infolge eines Maus- oder Tastaturereignisses ändert. Solche Änderungen werden über den Vergleich von *Delta.X* mit *HScrollBar^.Value* bzw. *Delta.Y* mit *VScrollBar^.Value* ermittelt. Vor der Neuausgabe wird der Cursor neu positioniert und die neu ermittelten Werte in *Delta* gespeichert. Siehe auch: *TView.DrawView*, *Value*, *HScrollBar*, *VScrollBar*.

ScrollTo `procedure ScrollTo(X, Y: Integer);`

Rollt die dargestellten Daten, bis die durch *X* und *Y* beschriebene Position im virtuellen Datenbereich erreicht ist, und veranlaßt – wenn notwendig – die neue Darstellung der Rollbalken. Siehe auch: *TView.DrawView*, *Value*, *HScrollBar*, *VScrollBar*.

SetLimit `procedure SetLimit(X, Y: Integer);`

Speichert die übergebenen Werte *X* und *Y* in *Delta* und ruft dann die *SetParam*s-Methode der verbundenen Rollbalken auf, die das Feld *Max* entsprechend der neuen Größe des Objekts ändert. Falls notwendig, wird das Objekt neu dargestellt. Siehe auch: *Limit*, *TScrollBar.SetParams*.

SetState
```
procedure SetState(AState: Word; Enable: Boolean);
                   virtual;
```

Diese Methode wird immer dann aufgerufen, wenn sich der

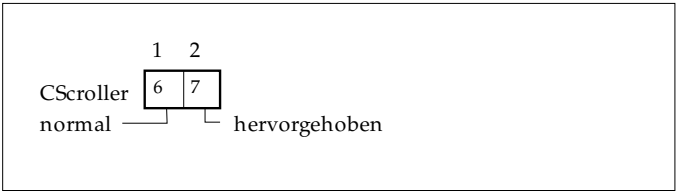
Status des Objekts ändert. Ist der neue Status *sfSelected* und *sfActive*, werden die Rollbalken dargestellt, andernfalls aus der Anzeige entfernt. Siehe auch: *TView.SetState*.

Store

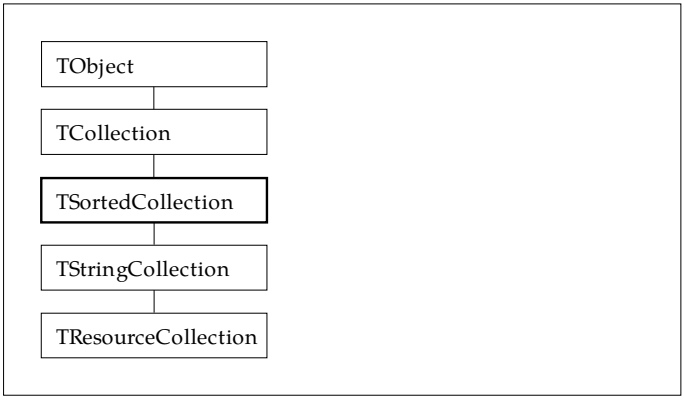
```
procedure Store(var S: TStream);
```

Schreibt ein *TSroller*-Objekt und – sofern vorhanden – die damit verbundenen Rollbalken in den angegebenen Stream. Siehe auch: *TScroller.Load*.

Farbpalette Dieses Objekt greift über die Palette *CScroller* auf die Einträge 6 und 7 der aktuellen Farbpalette des Programms zu.



TSortedCollection OBJECTS



TSortedCollection ist ein spezialisierter Nachkomme von *TCollection*, und stellt eine Kollektion bereit, deren Einträge nach einem bestimmten Schlüssel sortiert sind. Doppelte Einträge werden zurückgewiesen. Für die Sortierordnung verantwortlich ist die Methode *Compare*, eine abstrakte Methode, die Sie überschreiben müssen, um die Sortierordnung festzulegen, die Sie benötigen. Neue Elemente werden dann in der Reihenfolge eingefügt, die *Compare* bestimmt. Die Sortierschlüssel, die *Compare* vergleicht, stellt die Methode *KeyOf* bereit, die Sie ebenfalls überschreiben müssen. Ein Element können

Sie über *Search* finden; die Methode führt eine binäre Suche in der Kollektion durch.

Felder

Duplicates

```
Duplicates: Boolean;
```

Legt fest, ob die Kollektion mehrere Einträge mit dem gleichen Indexschlüssel aufnehmen soll und ist auf False gesetzt.

Methoden

Load

```
constructor Load(var S: TStream);
```

Inialisiert und lädt eine Kollektion über *TCollection.Load* aus dem angegebenen Stream. Dann wird auch der Wert für *Duplicates* aus dem Stream gelesen. Siehe auch: *TCollection.Load*.

Compare

```
function Compare(Key1, Key2: Pointer): Integer;
                 virtual;
```

Compare ist eine abstrakte Methode, muß also von Ihnen überschrieben werden. Ihre neue Methode muß folgende Werte zurückgeben: -1 wenn *Key1* < *Key2*, 0 wenn *Key1* = *Key2* und 1 wenn *Key1* > *Key2*. *Compare* wird auch von der Methode *Search* aufgerufen, um die binäre Suche durchzuführen. Siehe auch: *TSortedCollection.Search*.

IndexOf

```
function IndexOf(Item: Pointer): Integer; virtual;
```

Ruft die Methode *Search* auf und gibt, sofern vorhanden, den Index von *Item*^ zurück. Ist das Feld *Duplicates* True, ermittelt diese Methode über *Search* den ersten passenden Eintrag, und führt dann eine lineare Suche durch. Wurde *Item*^ nicht gefunden, gibt *IndexOf* -1 zurück. Siehe auch: *TSortedCollection.Search*.

Insert

```
procedure Insert(Item: Pointer); virtual;
```

Wenn das Element *Item*^ in der Kollektion noch nicht vorhanden ist, fügt *Insert* es an der richtigen Stelle ein. Die eigentliche Suche übernimmt die Methode *Search*, die auch die korrekte Einfügeposition ermittelt, wenn *Item*^ nicht gefunden wurde. Ist das Feld *Duplicates* True und wird der Eintrag vor eventuell vorhandenen Einträgen mit dem gleichen Schlüssel positioniert. Siehe auch: *TSortedCollection.Search*.

KeyOf

```
function KeyOf(Item: Pointer): Pointer; virtual;
```

Aufgabe dieser Methode ist, das Schlüsselfeld anzugeben, nach dem die Kollektion sortiert wird. Sofern nicht überschrieben, gibt die pseudo-abstrakte Methode *KeyOf* den Zeiger zurück, der über *Item* mitgegeben wurde; in der Regel

werden Sie also auch diese Methode überschreiben müssen. Siehe auch: *TSortedCollection.IndexOf*.

Search

```
function Search(Key: Pointer; var Index: Integer):
                Boolean; virtual;
```

Gibt *True* zurück, wenn das Element *Key^* in der Kollektion vorhanden ist. In diesem Fall enthält *Index* die Position des Eintrags, andernfalls die Position, die der Eintrag einnehmen müßte, würde er in die Kollektion eingefügt werden. Ist das Feld *Duplicates* True, gibt diese Methode den ersten Eintrag mit dem Schlüssel *Key^* zurück. Siehe auch: *TSortedCollection.Compare*, *TSortedCollection.Insert*.

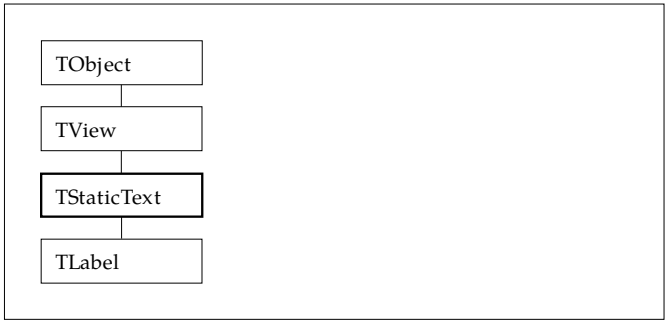
Store

```
procedure Store(var S: TStream);
```

Speichert die Kollektion einschließlich ihrer Einträge und dem Feld *Duplicates* im angegebenen Stream.

TStaticText

DIALOGS



TStaticText ist das einfachste aller in Turbo Vision definierten *View*-Objekte: es beschränkt sich darauf, einen Text auszugeben, und reagiert weder auf Tastatur- noch auf Mausereignisse.

Felder

Text

```
Text: PString;
```

Zeiger auf den String, der auszugeben ist.

Methoden

Init

```
constructor Init(var Bounds: TRect; AText: String);
```

Initialisiert über *TView.Init*(*Bounds*) ein *TStaticText*-Objekt in der in *Bounds* angegebenen Größe. Dem Feld *Text* wird ein Zeiger auf *AText* zugewiesen, der über *NewStr*(*AText*) ermittelt wird. Siehe auch: *TView.Init*.

Load `constructor Load(var S : TStream);`

Erstellt ein *TStaticText*-Objekt und initialisiert dessen Feld mit dem über *S.ReadStr* aus dem Stream gelesenen Wert. Diese Methode wird zusammen mit *TStaticText.Store* genutzt, um Objekte des Typs *TStaticText* in einem Stream zu speichern bzw. wieder zu laden. Siehe auch: *TView.Load*, *TView.Store*, *TStream.ReadStr*, *TStaticText.Store*.

Done `destructor Done; virtual;`

Gibt den *Text* zugeordneten Speicherplatz frei und ruft dann *TView.Done* auf. Siehe auch: *TView.Done*.

Draw `procedure Draw; virtual;`

Gibt den Text innerhalb der festgelegten Grenzen aus und umbricht ihn, wenn notwendig oder wenn ein *Ctrl-M* (Zeichen #13) innerhalb des Strings angetroffen wird. Beginnt der String mit *Ctrl-C* (Zeichen #3) wird er zentriert dargestellt.

GetPalette `function GetPalette: PPalette; virtual;`

Diese pseudo-abstrakte Methode gibt einen Zeiger auf die Farbpalette *CStaticText* zurück.

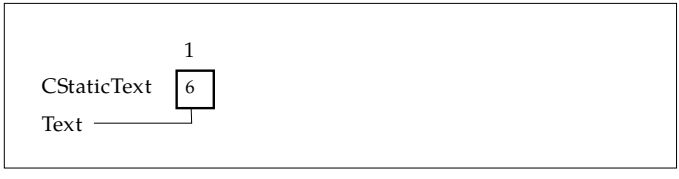
GetText `procedure GetText(var S: String); virtual;`

Gibt in *S* den String zurück, auf den *Text* zeigt.

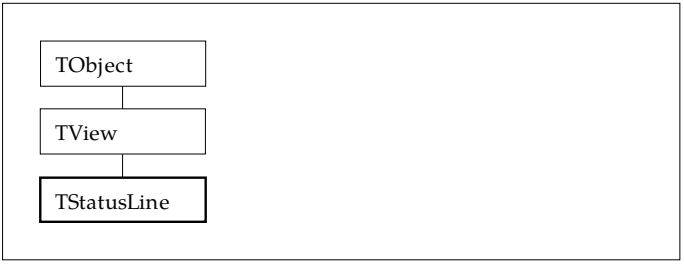
Store `procedure Store(var S: TStream);`

Speichert ein *TStaticText*-Objekt und dessen Feld *Text* über *S.WriteStr* in den angegebenen Stream. Diese Methode wird zusammen mit *TStaticText.Load* genutzt, um Objekte des Typs *TStaticText* in einem Stream zu speichern bzw. wieder zu laden. Siehe auch: *TView.Store*, *TStream.WriteStr*, *TStaticText.Load*.

Farbpalette Dieses Objekt greift über die Palette *CStaticText* auf den sechsten Eintrag der Palette *CDialog* zu.



TStatusLine



Dieses Objekt wird in der untersten Zeile des Bildschirms dargestellt. Sie können es zur Ausgabe von Informationen verwenden: Angaben über verfügbare Tastenkürzel oder Befehle, die dann auch mittels der Maus aktiviert werden können, den noch freien Arbeitsspeicher, die aktuelle Uhrzeit, Kurzinformationen zu Menüoptionen usw. Diese Einträge werden in einer verketteten Liste gespeichert und in Ihrem Programm über *InitStatusLine* ausgegeben. Wie die Menüleiste und das *TDesktop*-Objekt gehört auch die Statuszeile zu der Gruppe *TApplication*.

TStatusLine ist auch in der Lage, kurze Information zum aktuellen Programmkontext auszugeben. Dazu besitzt dieses Objekt das Feld *Defs*, ein Zeiger auf eine verkettete Liste von *TStatusDef*-Records, die einen Bereich von Hilfenummern und eine Liste von Statuszeilen-Einträgen speichern. Diese Einträge werden entsprechend dem aktuellen Programmkontext ausgegeben. Zu diesem Programmkontext kann darüber hinaus ein kurzer Informationstext dargestellt werden.

Felder

Item

```
Items: PStatusItem;
```

Ein Zeiger auf eine verkettete Liste von Records des Typs *TStatusItems*.

Defs

```
Defs: PStatusDef;
```

Ein Zeiger auf eine verkettete Liste von Records des Typs *TStatusDef*. Welche Liste verwendet wird, ist durch den Programmkontext festgelegt, der über den Wert von *HelpCtx* ermittelt wird. Siehe auch: *TStatusDef*, *TStatusLine.Update*, *TStatusLine.Hint*.

Methoden

Init

```
constructor Init(var Bounds: TRect; ADefs: PStatusDef);
```

Initialisiert über *TView.Init*(*Bounds*) eine Statuszeile in der in

Bounds angegebenen Größe. Das Feld *Options* wird auf *ofPreProcess*, das Feld *EventMask* auf *evBroadCast*, im Feld *GrowMode* werden die Bits *gfGrowLoY*, *gfGrowHiX* und *gfGrowHiY* gesetzt. *ADefs* wird im Feld *Defs* gespeichert; ist *ADefs* **nil**, wird auch Items auf **nil** gesetzt, andernfalls wird *Items* auf *ADefs^.Items* gesetzt. Siehe auch: *TView.Init*.

Load

```
constructor Load(var S: TStream);
```

Lädt ein *TStatusLine*-Objekt aus dem angegebenen Stream. Siehe auch: *TStatusLine.Store*.

Done

```
destructor Done; virtual;
```

Gibt den *Items* und *Defs* zugeordneten Speicherplatz frei und ruft dann *TView.Done* auf. Siehe auch: *TView.Done*.

Draw

```
procedure Draw; virtual;
```

Stellt die Statuszeile auf dem Bildschirm dar. Ist der über die Methode *Hint* ermittelte String nicht leer, wird zunächst ein Trennzeichen und dann der String selbst ausgegeben. Siehe auch: *TStatusLine.Hint*.

GetPalette

```
function GetPalette: PPalette;
```

Diese Methode gibt einen Zeiger auf die Palette *CStatusLine* zurück.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Behandelt alle der Statuszeile übergebenen Ereignisse. Handelt es sich dabei um einen Mausklick auf einen selektierbaren Eintrag, wird zunächst dessen Farbe geändert, dann die entsprechende Aktion ausgelöst. Dies gilt auch, wenn der Benutzer ein in der Statuszeile angezeigtes Tastenkürzel betätigt. Siehe auch: *TView.HandleEvent*.

Hint

```
function Hint(AHelpCtx: Word): String; virtual;
```

Diese pseudo-abstrakte Methode gibt einen leeren String zurück. Sie ist muß überschrieben werden, um einen String zurückzugeben, der eine Information zu dem durch *AHelpCtx* bezeichneten Programmkontext enthält. Diese Information wird dann, angeführt von einem Trennzeichen, in der Statuszeile ausgegeben. Siehe auch: *TStatusLine.Draw*.

Store

```
procedure Store(var S: TStream);
```

Schreibt über einen Aufruf von *TView.Store* ein *TStatusLine*-Objekt in den angegebenen Stream. Siehe auch: *TStatusLine.Load*.

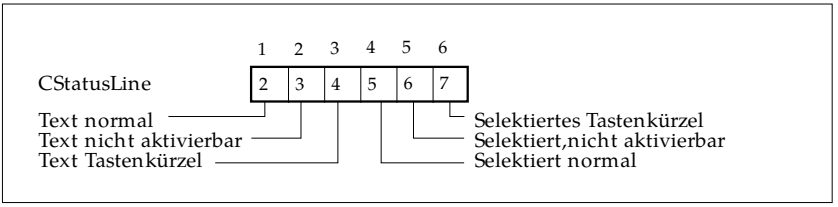
Update

```
procedure Update;
```

Stellt die Statuszeile neu dar, um eine Kurzinformation zum

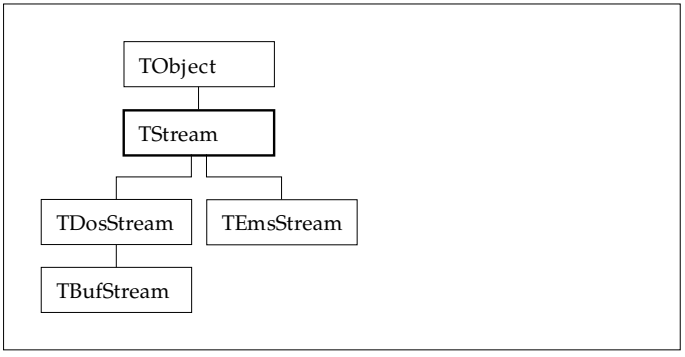
aktuellen Programmkontext auszugeben. Siehe auch: *TStatusLine.Defs*.

Farbpalette Dieses Objekt greift über die Palette *CStatusLine* auf die Einträge 2 bis 7 der aktuellen Farbpalette des Programms zu.



TStream

OBJECTS



TStream ist ein abstraktes Objekt, das grundlegende Methoden zur Ein- und Ausgabe polymorpher Objekte bereitstellt. Wenn Sie von diesem Objekt abgeleitete eigene *Stream*-Objekte definieren, müssen Sie die abstrakten Methoden *GetPos*, *GetSize*, *Read*, *Seek*, *Truncate* und *Write* überschreiben. Die in Turbo Vision definierten und von *TStream* abgeleiteten Objekte *TDosStream* und *TEmsStream* zeigen das dazu nötige Vorgehen. Wickelt Ihr *Stream*-Objekt die Ein- und Ausgabe über einen Puffer ab, müssen Sie zusätzlich auch die Methode *Flush* neu definieren.

Felder

Status

```
Status: Integer;
```

Dies Feld gibt an, ob ein Fehler aufgetreten ist oder nicht, und kann einen der folgenden Werte annehmen.

Enthält *Status* einen Wert ungleich *stOk* werden mit dem Stream solange keine weiteren Operationen durchgeführt, bis die Methode *Reset* aufgerufen wird.

Wert	Bedeutung
stOk	Kein Fehler aufgetreten
stError	Fehler beim Zugriff auf den *Stream*
stInitError	Fehler bei der Initialisierung des *Stream*
stReadError	Versuch, über das Ende des *Stream* zu lesen
stWriteError	Fehler beim Schreiben in den *Stream*
stGetError	Aufruf von *Get* mit einem nicht registrierten Objekttyp
stPutError	Aufruf von *Put* mit einem nicht registrierten Objekttyp

ErrorInfo

```
ErrorInfo: Integer;
```

Diese Feld enthält weitere Informationen, wenn *Status* nicht *stOk* ist. Enthält Status *stError*, *stInitError*, *stReadError* oder *stWriteError*, wird in diesem Feld – wenn verfügbar – der entsprechende DOS- oder EMS-Fehlercode ausgegeben. Ist *Status stGetError*, dann enthält *ErrorInfo* die Identifizierungsnummer des nicht registrierten Objekttyps (also den Wert des Feldes *ObjType* im *TStreamRec*). Ist *Status stPutError*, dann enthält *ErrorInfo* den Offset auf die VMT des nicht registrierten Objekttyps (also den Wert des Feldes *VmtLink* im *TStreamRec*).

Methoden

CopyFrom

```
procedure CopyFrom(var S: TStream; Count: LongInt);
```

Kopiert *Count* Bytes aus dem Stream *S* in den diese Methode aufrufenden Stream. Die folgenden Anweisungen zum Beispiel erstellen eine Kopie des gesamten Stream:

```
NewStream := New(TEmsStream, Init(OldStream^.GetSize));
                 OldStream.Seek(0);
NewStream^.CopyFrom(OldStream, OldStream^.GetSize);
```

Siehe auch: *TStream.GetSize*, *TObject.Init*.

Error

```
procedure Error(Code, Info: Integer); virtual;
```

Diese Methode wird immer augerufen, wenn ein Fehler auftrat. Sofern nicht überschrieben, beschränkt sie sich darauf, die Felder *Status* und *ErrorInfo* auf die Werte *Code* und *Info* zu setzen. Ist die globale Variable *StreamError* ungleich **nil**, wird die Prozedur aufgerufen, auf die diese Variable zeigt. Ist ein Fehler aufgetreten, werden alle Operationen mit dem Stream so lange nicht ausgeführt, bis *Reset* aufgerufen wird. Siehe auch: *ErrorCode*, *TStream.Reset*, *StreamError*.

Flush

```
procedure Flush; virtual;
```

Diese abstrakte Methode müssen Sie überschreiben, wenn Ihr *Stream*-Objekt Ein-/Ausgabe-Operationen über einen Puffer durchführt. Die Methode erzwingt dann bei einer Ausgabe-Operation das Schreiben des Pufferinhalts, bzw. löscht den Puffer im Lese-Modus. Siehe auch: *TDosStream.Flush*.

Get

```
function Get: PObject;
```

Gibt einen Zeiger auf das nächste Objekt im Stream zurück, also auf jenes Objekt, das von der aktuellen Position im Stream als nächstes verfügbar ist. Dazu ruft diese Methode über den *Load*-Zeiger des passenden *TStreamRec*-Records die Konstruktor-Methode *Load* dieses Objekts auf. Siehe auch: *TStream.GetPos*, *TStreamRec*, *stGetError*, *TStream.Put*, *RegisterType*.

GetPos

```
function GetPos: LongInt; virtual;
```

Diese abstrakte Methode ist dafür vorgesehen, die aktuelle Position innerhalb des Stream zurückzugeben, und muß überschrieben werden. Siehe auch: *TStream.Put*, *TStream.Seek*.

GetSize

```
function GetSize: LongInt; virtual;
```

Diese abstrakte Methode ist dafür vorgesehen, die Gesamtgröße des Stream zurückzugeben, und muß überschrieben werden. Siehe auch: *TStream.CopyFrom*.

Put

```
procedure Put(P: PObject);
```

Schreibt das Objekt *P^* an der aktuellen Position in den Stream. Dazu ruft diese Methode über den *Store*-Zeiger des passenden *TStreamRec*-Records die Methode *Store* dieses Objekts auf. Falls *P* gleich **nil** ist, schreibt ***Put*** ein Word mit dem Wert 0 in den Stream. Zeigt *P* auf ein nicht registriertes Objekt, wird *TStream.Error* aufgerufen und nichts in den Stream geschrieben. Siehe auch: *TStream.Get*, *TStreamRec*, *stPutError*, *TStream.Store*, *RegisterType*.

Read

```
procedure Read(var Buf; Count: Word); virtual;
```

Diese abstrakte Methode muß von allen unmittelbaren *TStream*-Nachkommen überschrieben werden. Sie kopiert, beginnend an der aktuellen Position im Stream, *Count* Bytes in *Buf*. Siehe auch: *TStream.Write*, *TStream.Error*.

ReadStr

```
function ReadStr: PString;
```

Liest an der aktuellen Position im Stream einen String ein und gibt einen Zeiger auf diesen String zurück. Über *GetMem* reserviert diese Methode für den gelesenen String einen Speicherbereich der Größe (Länge des String +1). Siehe auch: *TStream.WriteStr*, *stReadError*.

Reset
```
procedure Reset;
```
Setzt die Felder *ErrorInfo* und *Status* auf Null. Diese Methode ermöglicht nach dem Auftreten eines Fehlers und der Beseitigung von dessen Ursache die Weiterarbeit mit dem Stream. Siehe auch: *ErrorInfo*, *TStream.Error*, *TStream.Status*.

Seek
```
procedure Seek(Pos: LongInt); virtual;
```
Diese abstrakte Methode muß von allen unmittelbaren *TStream*-Nachkommen überschrieben werden. Sie setzt die aktuelle Position auf die Stelle *Pos* Bytes ab Beginn des Stream. Der Stream beginnt mit der Position 0. Siehe auch: *TStream.GetPos*.

Truncate
```
procedure Truncate; virtual;
```
Diese abstrakte Methode muß von allen unmittelbaren *TStream*-Nachkommen überschrieben werden. Sie löscht ab der aktuellen Position bis zum Ende des Stream alle Daten. Die Position selbst verändert sich nicht, zeigt also nach der Operation auf das Ende des Stream. Siehe auch: *TStream.GetPos*, *TStream.Seek*.

Write
```
procedure Write(var Buf; Count: Word); virtual;
```
Diese abstrakte Methode muß von allen unmittelbaren *TStream*-Nachkommen überschrieben werden. Sie schreibt, beginnend an der aktuellen Position im Stream, *Count* Bytes aus *Buf* in den Stream. Siehe auch: *TStream.Read*, *TStream.GetPos*, *TStream.Seek*, *TStream.Error*.

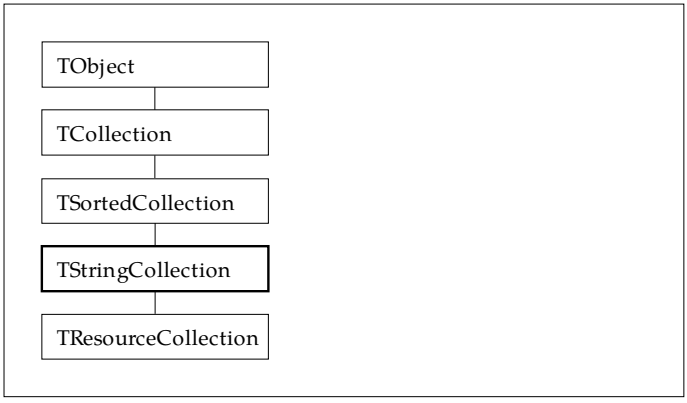
WriteStr
```
procedure WriteStr(P: PString);
```
Schreibt des String *P^* an der aktuellen Position in den Stream. Siehe auch: *TStream.ReadStr*.

TStringCollection OBJECTS

TStringCollection ist ein Nachkomme von *TSortedCollection* und stellt eine Kollektion für die Speicherung von Strings bereit. Die *Compare*-Methode von *TSortedCollection* wird überschrieben, um die lexikalisch sortierte Anordnung von Zeichenketten zu gewährleisten.

Methoden

Compare
```
function Compare(Key1, Key2: Pointer): Integer;
                 virtual;
```
Vergleicht die Strings *Key1^* und *Key2^* miteinander und gibt einen der folgenden Werte zurück: -1 wenn *Key1* < *Key2*, 0



wenn *Key1* = *Key2* und +1 wenn *Key1* > *Key2*. Siehe auch: *TStringCollection.Search*.

FreeItem

```
procedure FreeItem(Item: Pointer); virtual;
```

Entfernt den String *Item*^ aus der Kollektion.

GetItem

```
function GetItem(var S: TStream): Pointer; virtual;
```

Liest, sofern nicht überschrieben, einen Eintrag aus *TStream*, und benutzt dazu die Methode *S.ReadStr*. Siehe auch: *TStream.ReadStr*.

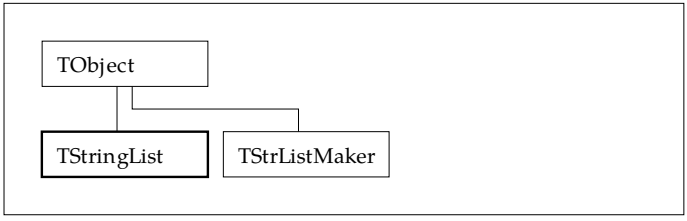
PutItem

```
procedure PutItem(var S: TStream; Item: Pointer);
                  virtual;
```

Übergibt, sofern nicht überschrieben, den Eintrag *Item*^ an *TStream*, und benutzt dazu die Methode *S.WriteStr*. Siehe auch: *TStream.WriteStr*.

TStringList

OBJECTS



TStringList stellt eine spezielle Ressourcen-Datei zur Verfügung, die Strings speichert, die über eine eindeutige Nummer (0 bis 65535) angesprochen werden. Dies erlaubt Ihnen zum Beispiel, Ihr Programm relativ einfach auf eine andere Lan-

dessprache umzustellen, da dazu nur diese Strings zu ändern sind. *TStringList* definiert nur Methoden, die das Lesen dieser Strings erlauben. Benutzen Sie das Objekt *TStrListMaker*, um eine solche Ressourcen-Datei zu erstellen.

Beachten Sie, daß *TStringList* und *TStringListMaker* dieselbe Identifikationsnummer (Feld *ObjType* im *TStreamRec*-Record) zugewiesen ist und daß diese daher nicht im selben Programm verwendet werden können.

Methoden

Load

```
constructor Load(var S: TStream);
```

Liest den Index der Stringliste aus dem angegebenen Stream und speichert eine Referenz auf diesen Stream, um später Strings einlesen zu können.

Die folgenden Anweisungen zeigen, wie Sie ein *TStringList*-Objekt instantiieren. Voraussetzung ist, daß dieses Objekt via *RegisterType*(*RStringList*) angemeldet wurde. Die Ressourcen-Datei wurde vorher über *TStringListMaker* und der Methode *TResourceFile.Put* erzeugt.

```
ResFile.Init(New(TBufStream, Init('MYAPP.RES',
                                  stOpenRead, 1024));
Strings := PStringList(ResFile.Get('Strings'));
```

Siehe auch: *TStrListMaker.Init*, *TStringList.Get*.

Done

```
destructor Done; virtual;
```

Gibt den von diesem Objekt belegten Speicher frei. Siehe auch: *TStrListMaker.Init*.

Get

```
function Get(Key: Word): String;
```

Gibt den String mit der Nummer *Key* oder, wenn dieser Eintrag nicht vorhanden ist, einen leeren String zurück. Beispiel:

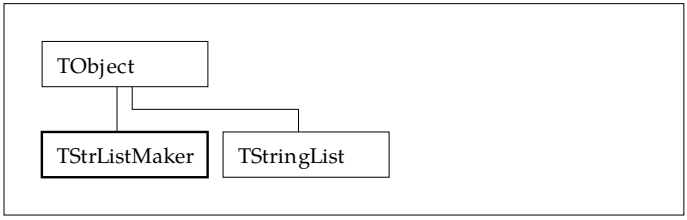
```
P := @FileName;
FormatStr(S, Strings^.Get(sLoadingFile); P);
```

Siehe auch: *TStrListMaker.Put*.

TStrListMaker OBJECTS

TStrListMaker ist ein einfaches Objekt, dessen Aufgabe darin besteht, Objekte des Typs *TStringList* zu erstellen und zu verwalten.

Der folgende Programmausschnitt zeigt, wie Sie eine Ressourcen-Datei anlegen und Strings in ihr speichern können:




```
const
  sInformation = 100;
  sWarning     = 101;
  sError       = 102;
  sLoadingFile = 200;
  sSavingFile  = 201;

var
  ResFile: TResourceFile;
  S: TStrListMaker;

begin
  RegisterType(RStringListMaker);

  ResFile.Init(New(TBufStream, Init('MYAPP.RES',
               stCreate, 1024)));
  S.Init(16384, 256);

  S.Put(sInformation, 'Information');
  S.Put(sWarning, 'Warnung');
  S.Put(sError, 'Fehler');
  S.Put(sLoadingFile, 'Datei %s wird geladen.');
  S.Put(sSavingFile, 'Datei %s wird gesichert.');

  ResFile.Put(@S, 'Strings');
  S.Done;
  ResFile.Done;
end;
```

Methoden

Init `constructor Init(AStrSize, AIndexSize: Word)`

Erstellt im Speicher eine Liste von Strings in der über *AStrSize* angegebenen Größe und einen Index, der *AIndexSize* Elemente aufnehmen kann, und reserviert für die beiden Puffer den notwendigen Speicher.

AStrSize muß groß genug gewählt werden, um alle Strings, die in die Liste eingefügt werden, aufnehmen zu können. Jeder String belegt *Length*(*Str*) + *1* Byte Speicher.

Wenn Sie über *TStrListMaker.Put* einen String in die Liste einfügen, wird auch ein entsprechender Index-Eintrag erzeugt. Indexeinträge für Strings, die über eine fortlaufende

Nummer angesprochen werden (wie im obigen Beipiel *sInformation*, *sWarnings* und *sError*), werden zusammen in einem Indexrecord gespeichert. Ein solcher Record kann bis zu 16 Einträge aufnehmen, wobei jeder Eintrag 6 Bytes belegt. *AStrIndex* muß groß genug gewählt werden, um alle notwendigen Indexrecords aufnehmen zu können. Siehe auch: *TStringList.Load*, *TStrListMaker.Done*.

Done `destructor Done; virtual;`

Gibt den von diesem Objekt belegten Speicher frei. Siehe auch: *TStrListMaker.Init*.

Put `procedure Put(Key: Word; S: String);`

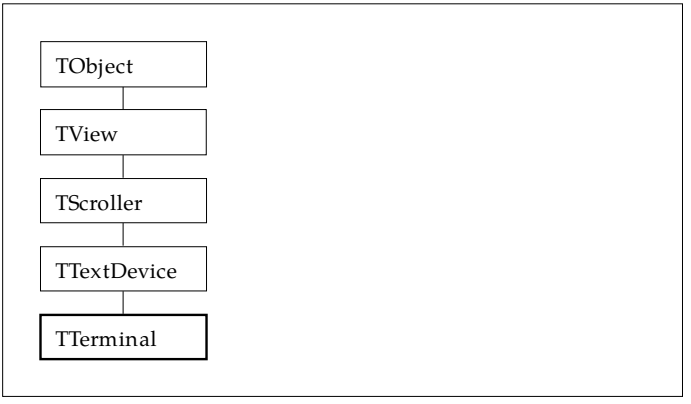
Fügt den übergebenen String *S* in der Liste und den Schlüssel *Key* in den Index ein.

Store `procedure Store(var S: TStream);`

Schreibt die Liste der Strings in den angegebenen Stream.

TTerminal

TEXTVIEW



TTerminal implementiert ein einfaches Terminal, über das Strings gelesen und geschrieben werden können. Diese Operationen laufen über einen Ringpuffer, der gemäß Voreinstellung 64 KBytes groß ist.

Felder

BufSize `BufSize: Word;`

Die Größe des Terminal-Puffers in Bytes.

Buffer `Buffer: PTerminalBuffer;`

Zeiger auf den dem Puffer zugeordneten Speicherbereich.

QueFront `QueFront: Word;`

Byte-Offset auf das erste Zeichen im Puffer.

QueBack `QueBack: Word;`

Byte-Offset auf das letzte Zeichen im Puffer.

Methoden

Init
```
constructor Init(var Bounds: TRect; AHScrollBar,
                 AVScrollBar: PScrollBar; ABufSize:
                 Word);
```

Initialisiert ein *TTerminal*-Objekt von der in *Bounds* angegebenen Größe, versehen mit einem horizontalen und vertikalen Rollbalken, und reserviert *ABufSize* Speicher für den Puffer. *ABufSize* wird in *BufSize* gespeichert. *QueFront* und *QueBack* werden auf 0 gesetzt, was anzeigt, daß der Puffer leer ist. In *GrowMode* werden die Bits *gfGrowHiX* und *gfGrowHiY* gesetzt. Der Cursor wird an der Position 0, 0 – dem Ursprung des *View*-Objekts – ausgegeben. Siehe auch: *TScroller.Init.*

Done `destructor Done; virtual;`

Gibt den von dem Puffer belegten Speicher frei und ruft dann *TTextDevice.Done* zur Freigabe des Objekts auf. Siehe auch: *TScroller.Done*, *TTextDevice.Done*.

BufDec `procedure BufDec(var Val: Word);`

Wird benutzt, um einen Offsetwert im Puffer herunterzusetzen, eine Operation, die berücksichtigen muß, daß dieser Puffer als Ringpuffer definiert ist. Hat *Val* den Wert 0, wird *Val* auf *BufSize – 1* gesetzt, andernfalls wird *Val* dekrementiert.

BufInc `procedure BufInc(var Val: Word);`

Wird benutzt, um einen Offsetwert im Puffer heraufzusetzen, eine Operation, die berücksichtigen muß, daß dieser Puffer als Ringpuffer definiert ist. *Val* wird inkrementiert; hat *Val* darauf einen Wert >= *BufSize*, wird *Val* auf 0 gesetzt.

CalcWidth `function CalcWidth: Integer;`

Gibt die Länge der längsten Zeile im Puffer zurück.

CanInsert `function CanInsert(Amount: Word): Boolean;`

Gibt True zurück, wenn die in *Amount* angegebene Anzahl Bytes in den Terminal-Puffer eingefügt werden kann, ohne die Daten am Anfang des Puffers zu überschreiben.

Draw `procedure Draw; virtual;`

Wird immer dann aufgerufen, wenn das Objekt neu dargestellt werden muß, wenn also die Rollbalken aktiviert wurden, die Größe des Objekts sich änderte oder weiterer Text darzustellen ist.

NextLine

```
function NextLine(Pos: Word): Word;
```

Sucht im Puffer ab der Position *Pos* in Richtung auf das Ende des Puffers nach dem Beginn einer neuen Zeile und gibt einen Offset auf diesen Zeilenbeginn zurück. Siehe auch: *TTerminal.PrevLines.*

PrevLines

```
function PrevLines(Pos: Word; Lines: Word): Word;
```

Sucht im Puffer ab der Position *Pos* in Richtung auf den Anfang des Puffers nach dem Beginn einer neuen Zeile, die *Lines* Zeilen von der Position *Pos* entfernt ist und gibt einen Offset auf diesen Zeilenbeginn zurück. Siehe auch: *TTerminal.NextLine.*

StrRead

```
function StrRead(var S: TextBuf): Byte; virtual;
```

Diese Methode beschränkt sich darauf, 0 zurückzugeben. Sie ist gedacht, Text von einer Text-Ein-/Ausgabeeinheit einzulesen und muß, soll sie diese Funktion erfüllen, überschrieben werden.

StrWrite

```
procedure StrWrite(var S: TextBuf; Count: Byte);
                   virtual;
```

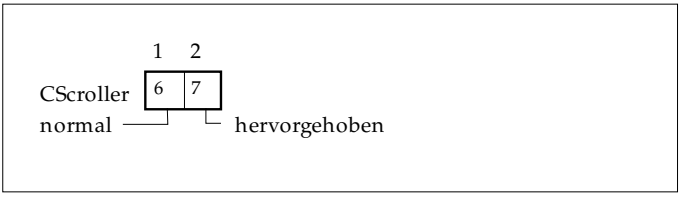
Fügt bis zu *Count* Zeichen Text aus *S* in den Puffer des Terminals ein. Diese Methode veranlaßt auch – wenn durch diese Einfügung notwendig geworden – über *DrawView* eine neue Darstellung des Objekts. Siehe auch: *TView.DrawView.*

QueEmpty

```
function QueEmpty: Boolean;
```

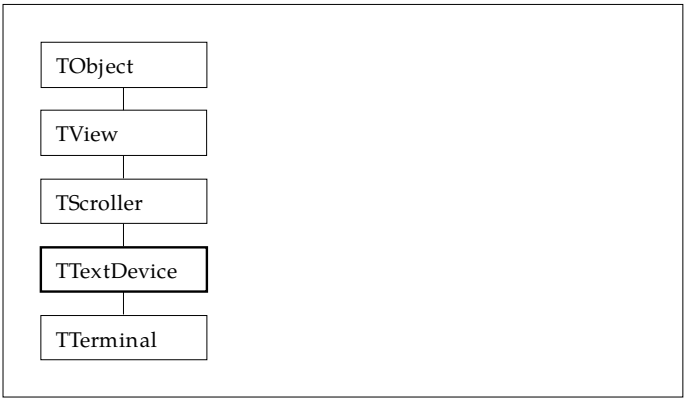
Gibt True zurück, wenn *QueFront* gleich *QueBack*, der Puffer also leer ist.

Farbpalette Dieses Objekt greift über die Palette *CScroller* auf die Einträge 6 und 7 der aktuellen Farbpalette des Programms zu.



TTextDevice

TEXTVIEW



TTextDevice nutzt die von seinem Vorfahren *TScroller* geerbten Fähigkeiten, um einen einfaches TTY-ähnliches, textorientiertes Ein-/Ausgabegerät zu implementieren. Die virtuellen Methoden, die dieses Objekt definiert, dienen dazu, Strings einzulesen und auszugeben. *TTextDevice* stellt lediglich ein Basis-Objekt für die Ableitung von Terminal-Objekten dar.

Methoden

StrRead

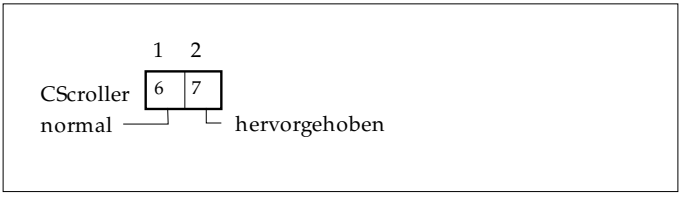
```
function StrRead(var S: TextBuf): Byte; virtual;
```

Diese Methode beschränkt sich darauf, 0 zurückzugeben. Sie ist gedacht, Text von einer Text-Ein-/Ausgabeeinheit einzulesen, und muß, soll sie diese Funktion erfüllen, überschrieben werden.

StrWrite

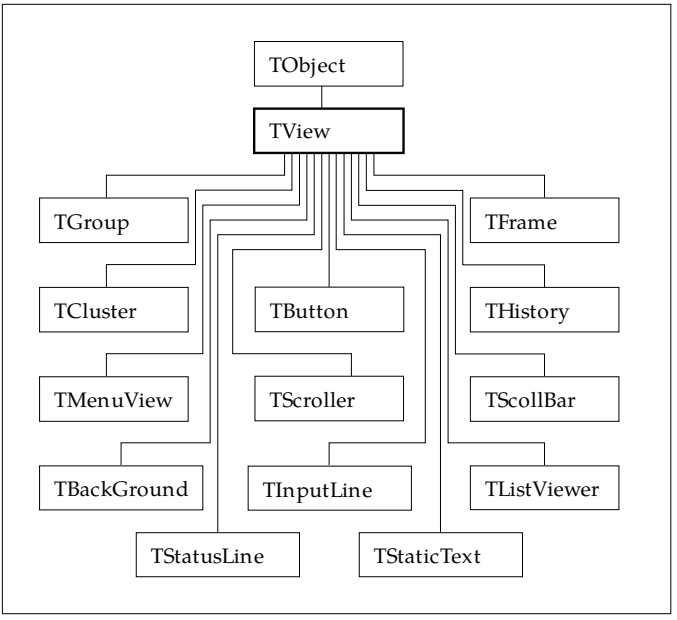
```
procedure StrWrite(var S: TextBuf; Count: Byte);
                   virtual;
```

Diese abstrakte Methode ist dazu gedacht, der Text-Ein-/Ausgabeeinheit Strings zu übergeben. *TTerminal.StrWrite* zum Beispiel fügt bis zu *Count* Zeilen Text aus *S* in den Puffer des Terminals ein. Siehe auch: *TTerminal.StrWrite*.



Farbpalette Dieses Objekt greift über die Palette *CScroller* auf die Einträge 6 und 7 der aktuellen Farbpalette des Programms zu.

TView VIEWS



Ein Turbo-Vision-Programm, das dieses Objekt nicht einbindet, ist kaum vorstellbar. In der Regel werden Sie jedoch nicht dieses Objekt, sondern seine zahlreichen Nachfolger instantiieren, für die *TView* die grundlegenden Felder und Methoden bereitstellt.

Felder

Owner

```
Owner: PGroup;
```

Ist auf **nil** gesetzt, wenn das Objekt zu keiner Gruppe gehört, andernfalls zeigt *Owner* auf das *TGroup*-Objekt, dem es angehört. Es wird dann innerhalb des Rechtecks, das von der Gruppe verwaltet wird, dargestellt und gegegebenfalls an dessen Grenzen abgeschnitten.

Next

```
Next: PView;
```

Ein Zeiger auf die in der Z-Ordnung nächste Subview der Gruppe, zu der dieses Objekt gehört. Wenn dieses Objekt selbst die letzte Subview ist, zeigt *Next* auf die erste zu der Gruppe gehörenden Subview.

Origin `Origin: TPoint;`

Speichert die Koordinaten (X und Y) der oberen linken Ecke des Objekts. Diese Werte beziehen sich auf den Ursprung des Koordinatensystems der Gruppe, zu der das Objekt gehört. Siehe auch: *MoveTo*, *Locate*.

Size `Size: TPoint;`

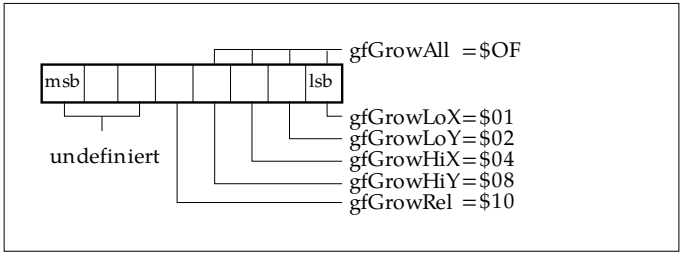
Speichert die Breite und die Höhe des *TView*-Objekts. Siehe auch: *TView.DataSize*, *TView.ChangeBounds*, *TView.GrowTo*.

Cursor `Cursor: TPoint;`

Die Position des Cursors innerhalb des Objekts. Sichtbar ist der Cursor nur dann, wenn dieses Objekt fokussiert (*sfFocused*) und über *sfCursorVis* aktiviert ist. Er wird entweder in seiner Standardform als Unterstrich oder als voller Block dargestellt, und kann über *sfCursorIns* in seiner Form verändert werden. Siehe auch: *TView.SetCursor*, *Normal-Cursor, Blockcursor*.

GrowMode `GrowMode: Byte;`

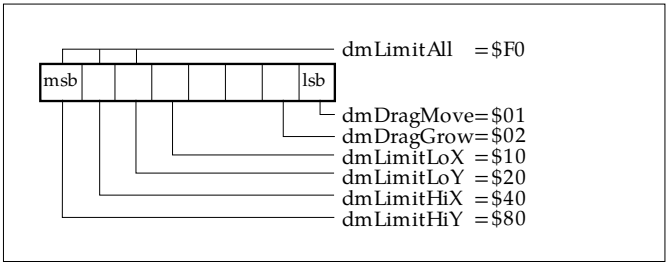
Legt fest, wie sich die Größe des Objekts verändert, wenn die Größe der Gruppe, zu der es gehört, geändert wird. Die einzelnen Bits haben folgende Bedeutung:



Beispiel: `GrowMode := gfGrowLoX or gfGrowLoY;`

Siehe auch: *gfXXXX*-Konstanten.

DragMode `DragMode: Byte;`



Legt das Verhalten des Objekts fest, wenn es über den Bildschirm bewegt wird. Die einzelnen Bits haben die in der Grafik dargestellte Bedeutung.

Siehe auch: *TView.DragView*, *dmXXXX*-Konstanten.

HelpCtx

```
HelpCtx: Word;
```

Die Hilfenummer dieses Objekts. Ist das *TView*-Objekt fokussiert, dann können Sie über *GetHlpCtx* diese Nummer abfragen, um zu erfahren, aus welchem Programmkontext heraus der Benutzer Hilfe angefordert hat. Siehe auch: *TView.GetHlpCtx*.

State

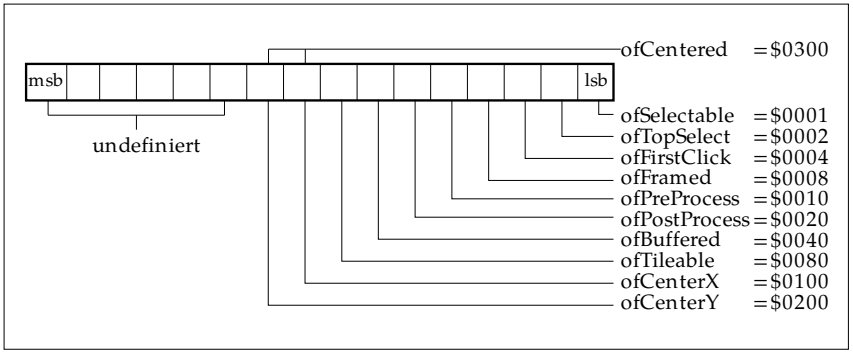
```
State: Word;
```

Die einzelnen Bits von *State* repräsentieren den aktuellen Zustand des Objekts. Viele Methoden, die *TView* definiert, überprüfen über *GetState*(*AState*) den aktuellen Status (diese Funktion gibt True zurück, wenn *State* dem Parameter *AState* entspricht) und verändern ihn über *SetState*. Die einzelnen Bits von *State* können mittels der *sfXXXX*-Konstanten gesetzt oder gelöscht werden.

Options

```
Options: Word;
```

Die einzelnen Bits von *Options* bestimmen verschiedene Eigenschaften des *View*-Objekts und haben folgende Bedeutung:



Siehe auch: *ofXXXX*-Konstanten.

EventMask

```
EventMask: Word;
```

Die in *EventMask* gesetzten Bits bestimmen, für welche Arten von Ereignissen das *View*-Objekt zuständig ist. Das bedeutet umgekehrt: Ist ein Bit nicht gesetzt, werden die darüber spezifizierten Ereignisse ignoriert. Wird *EventMask* der Wert $FFFF zugewiesen, ist das Objekt für sämtliche Ereignisse zuständig; bei Zuweisung von 0 werden keine Ereignisse

bearbeitet. Den einzelnen Bitpositionen entsprechen in Turbo Vision definierte Konstanten: *evNothing*, *evMouseDown*, *evMouseUp*, *evMouseAuto*, *evKeyDown*, *evCommand*, *evBroadCast* und *evMessage*. Bei seiner Initialisierung wird die Zuständigkeit des *View*-Objekts auf die Ereignisse *evMouseDown*, *evKeyDown* und *evCommand* beschränkt. Siehe auch: *TView.HandleEvent*.

Methoden

Init

```
constructor Init(var Bounds: TRect);
```

Initialisiert ein *TView*-Objekt, dessen Größe über das Argument *Bounds* festgelegt ist, und belegt die Felder mit den folgenden Werten:

Owner	**nil**
Next	**nil**
Origin	(Bounds.A.X, Bounds.A.Y)
Size	(Bounds.B.X - Bounds.A.X, Bounds.B.Y - Bounds.A.Y)
Cursor	(0, 0)
GrowMode	0
DragMode	*dmLimitLoY*
HelpCtx	*hcNoContext*
State	*sfVisible*
Options	0
EventMask	*evMouseDown* + *evKeyDown* + *evCommand*

Beachten Sie, daß *TObject.Init* die Felder aller Nachkommen von *TView* löscht. Rufen Sie daher *TView.Init* auf, *bevor* Sie diese Felder mit Werten belegen. Siehe auch: *Object.Init*

Load

```
constructor Load(var S: TStream);
```

Initialisiert ein *TView*-Objekt mit den Feldwerten, die aus *S* eingelesen werden. Siehe auch: *TView.Store*, *TStream.Get*.

Done

```
destructor Done; virtual;
```

Nimmt das Objekt vom Bildschirm und entfernt es, wenn es zu einer Gruppe gehört, aus dieser Gruppe.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Diese Methode rufen Sie auf, wenn das *TView*-Objekt das Ereignis in *Event* abarbeiten soll. *HandleEvent* signalisiert seinerseits die erfolgreiche Bearbeitung des Ereignisses über einen Aufruf der Methode *ClearEvent*. So wie sie ist, kann diese Methode nur auf *evMouseDown*-Ereignisse reagieren: Ist das Objekt noch nicht selektiert (*sfSelected*), aber selektierbar (*sfSelectable*) und in State das Bit *sfDisabled* nicht gesetzt, selektiert sich das Objekt über einen Aufruf von *Select*. Siehe auch: *TView.ClearEvent*.

BlockCursor

```
procedure BlockCursor;
```

Setzt das Bit *sfCursorIns* in *State*, damit der Cursor als voller Block erscheint. Allerdings ist der Cursor nur dann sichtbar, wenn auch *sfCursorVis* gesetzt und darüber hinaus auch das *TView*-Objekt selbst sichtbar ist. Siehe auch: *sfCursorIns*, *sfCursorVis*, *TView.NormalCursor*, *TView.ShowCursor*, *TView.HideCursor*.

CalcBounds

```
procedure CalcBounds(var Bounds: TRect; Delta:
                     TPoint); virtual;
```

Berechnet die Position und die Größe des Objekts neu, wobei angenommen wird, daß die Größe der Gruppe, der das Objekt angehört, entsprechend den Werten in *Delta* geändert wurde. Die neuen Werte werden einerseits in *Origin* und *Size* festgehalten, aber auch über *Bounds* zurückgegeben. Siehe auch: *TView.GetBounds*, *TView.ChangeBounds*.

ChangeBounds

```
procedure ChangeBounds(var Bounds: TRect); virtual;
```

Ändert die Position (*Origin*) und Größe (*Size*) des Objekts auf die in *Bounds* übergebenen Werte. Diese Methode wird in der Regel von anderen Methoden und nicht direkt aufgerufen. Siehe auch: *TView.Locate*, *TView.MoveTo*.

ClearEvent

```
procedure ClearEvent(var Event: TEvent);
```

Diese Methode wird von *HandleEvent* aufgerufen, um anzuzeigen, daß ein Ereignis erfolgreich abgearbeitet wurde. *ClearEvent* setzt *Event.What* auf *evNothing* und *Event.InfoPtr* auf *@Self*. Siehe auch: *TView.HandleEvent*.

CommandEnabled

```
function CommandEnabled(Command: Word): Boolean;
```

Gibt True zurück, wenn der in *Command* angegebene Befehle aktivierbar ist, andernfalls False. Wenn Sie die Modalität des *TView*-Objekts ändern, können Sie jeden der verfügbaren Befehle als aktivierbar oder nicht aktivierbar definieren. Ändern Sie die Modalität erneut, werden auch die Befehle in den vorher gültigen Zustand zurückversetzt. Siehe auch: *TView.DisableCommands*, *TView.EnableCommands*, *TView.SetCommands*.

DataSize

```
function DataSize: Word; virtual;
```

Eine abstrakte Methode, die Sie überschreiben müssen. Sie gibt die Größe der Daten zurück, die *GetData* aus bzw. *SetData* in den Datenrecord kopiert. Diese Methode werden Sie bei der Initialisierung von Gruppen verwenden müssen. Siehe auch: *TView.GetData*, *TView.SetData*.

DisableCommands

```
procedure DisableCommands(Commands: TCommandSet);
```

Definiert die in *Commands* angegebenen Befehle als nicht aktivierbar. Siehe auch: *TView.CommandEnabled*, *TView.EnableCommands*, *TView.SetCommands*.

DragView

```
procedure DragView(Event: TEvent; Mode: Byte;
                   var Limits: TRect; MinSize,
                   MaxSize: TPoint);
```

Bewegt das *TView*-Objekt innerhalb der in *Limits* angegebenen Grenzen über den Bildschirm, wobei das Feld *DragMode* auf *Mode* gesetzt wird und die Bewegung steuert. Über *MaxSize* und *MinSize* können Sie angeben, welche maximale und welche minimale Größe das Objekt nicht überschreiten darf. Das Ereignis, das den Aufruf von *DragView* bewirkt, wird in *Event* übergeben, damit zwischen Positionsveränderung mithilfe der Maus und mittels der Cursortasten unterschieden werden kann.

Draw

```
procedure Draw; virtual;
```

Wird immer dann aufgerufen, wenn das Objekt sich auf dem Bildschirm darstellen muß. *Draw* ist dafür verantwortlich, daß der gesamte Bereich des Bildschirms, für den das Objekt verantwortlich ist, neu aufgebaut wird. Daher muß diese Methode für von *TView* abgeleitete Objekte überschrieben werden. *Draw* wird selten direkt aufgerufen, da *DrawView* wesentlich effektiver arbeitet; diese Methode ruft *Draw* nur dann auf, wenn das Objekt tatsächlich auf dem Bildschirm sichtbar ist. Siehe auch: *TView.DrawView*.

DrawView

```
procedure DrawView;
```

Diese Methode ruft *Draw* auf, allerdings nur dann, wenn das Objekt tatsächlich auf dem Bildschirm sichtbar ist. Daher sollten Sie, wenn das Objekt neu auszugeben ist, diese Methode und nicht *Draw* aufrufen. Siehe auch: *TView.Draw*, *TGroup.DrawSubViews*, *TView.Exposed*.

EnableComands

```
procedure EnableCommands(Commands: TCommandSet);
```

Definiert die in *Commands* angegebenen Befehle als aktivierbar. Siehe auch: *TView.DisableCommands*, *TView.GetCommands*, *TView.CommandEnabled*, *TView.SetCommands*.

EndModal

```
procedure EndModal(Command: Word); virtual;
```

Das *TView*-Objekt verliert die Eigenschaft der Modalität. *Command* wird der Methode *ExecView* (die dieser Gruppe die Modalität zugeordnet hat) übergeben, die ihrerseits *Command* als Funktionsresultat zurückgibt.

EventAvail

```
function EventAvail: Boolean;
```

Gibt True zurück, wenn ein Ereignis zur Übergabe an *Handle-*

Event verfügbar ist. Siehe auch: *TView.MouseEvent*, *TView.KeyEvent*, *GetEvent*.

Execute

```
function Execute: Word; virtual;
```

Diese abstrakte Methode wird immer dann aufgerufen, wenn das Objekt als modal definiert wird. Sie muß überschrieben werden. Ihr Zweck besteht darin, in einer Schleife Ereignisse entgegenzunehmen, weiterzugeben und ihre Bearbeitung zu kontrollieren. Allerdings wird diese Methode niemals direkt aufgerufen.

Exposed

```
function Exposed: Boolean;
```

Gibt True zurück, wenn das *TView*-Objekt, und sei es auch nur zum Teil, auf dem Bildschirm sichtbar ist. Siehe auch: *sfExposed*, *TView.DrawView*.

GetBounds

```
procedure GetBounds(var Bounds: TRect);
```

Gibt über *Bounds* die Dimension des Objekts, bezogen auf das Koordinatensystem der Gruppe, zu der es gehört, zurück. Siehe auch: *Origin*, *Size*, *TView.CalcBounds*, *TView.ChangeBounds*.

GetClipRect

```
procedure GetClipRect(var Clip: TRect);
```

Gibt über *Clip* die Größe des Rechtecks zurück, das von diesem Objekt tatsächlich auf dem Bildschirm sichtbar ist. Diese Methode wird von *Draw* aufgerufen, um die Ausgabe des Objekts auf dem Bildschirm zu beschleunigen. Siehe auch: *TView.View*.

GetColor

```
function GetColor(Color: Word): Word;
```

Diese Methode gibt die tatsächlich zu verwendenden Farbattribute zurück, indem sie das nieder- und das höherwertige Byte in *Color* als Indices in die Farbpalette des *TView*-Objekts und in die Farbpalette der Gruppe, zu der es gehört, verwendet. Siehe auch: *TView.GetPalette*.

GetCommands

```
procedure GetCommands(var Commands: TCommandSet);
```

Gibt über das Argument *Commands* die verfügbaren Befehle zurück. Siehe auch: *TView.CommandsEnabled*, *TView.EnableCommands*, *TView.DisableCommands*, *TView.SetCommands*.

GetData

```
procedure GetData(var Rec); virtual;
```

Abstrakte Methode, die Daten in den angegebenen *Rec* schreibt. Siehe auch: *TView.SetData*, *TGroup.SetData*.

GetEvent

```
procedure GetEvent(var Event: TEvent); virtual;
```

Gibt das nächste verfügbare Ereignis über *Event* zurück oder *evNothing*, wenn kein Ereignis ansteht. Sofern nicht über-

schrieben, ruft diese Methode die *GetEvent*-Methode der Gruppe auf, der das Objekt angehört. Siehe auch: *TView.EventAvail*, *TProgram.Idle*, *TView.HandleEvent*, *TView.PutEvent*.

GetExtent

```
procedure GetExtent(var Extent: TRect);
```

Gibt über *Extent* die Breite und die Höhe des Objekts zurück, also die Werte, die im Feld *Size* gespeichert sind. Siehe auch: *Size*, *Origin*, *TView.CalcBounds*, *TView.GetBounds*.

GetHelpCtx

```
function GetHelpCtx: Word; virtual;
```

Gibt den in dem Feld *HelpCtx* gespeicherten Wert zurück. Über diesen Wert erfährt ein Hilfesystem, aus welchem Programmkontext heraus der Benutzer Hilfe angefordert hat. Siehe auch: *HelpCtx*, *sfFocused*.

GetPalette

```
function GetPalette: PPalette; virtual;
```

GetPalette muß einen Zeiger auf die Farbpalette des Objekts zurückgeben oder **nil**, wenn keine Farbpalette vorhanden ist. Diese Methode wird von *GetColor*, *WriteChar* und *WriteStr* aufgerufen, um Indices in eine Farbpalette in verwendbare Bildschirmattribute zu konvertieren. Ein Wert von **nil** bewirkt, daß keine Konvertierung vorgenommen wird. *GetPalette* gibt **nil** zurück und wird von abgeleiteten Objekten fast immer überschrieben. Siehe auch: *TView.GetColor*, *TView.WriteXXX*.

GetPeerViewPtr

```
procedure GetPeerViewPtr(var S: TStream; var P);
```

Lädt eine Referenz auf ein Objekt, das zur gleichen Gruppe wie das aufrufende Objekt gehört, aus dem Stream und gibt in *P* einen Zeiger auf dieses Objekt zurück. Dieser Zeiger ist aber erst dann gültig, wenn alle zur Gruppe gehörenden Objekte aus dem Stream geladen wurden. *GetViewPeerPtr* darf nur innerhalb eines *Load*-Konstruktors aufgerufen werden und dient dazu, Zeiger auf Objekte, die über *PutPeerViewPtr* innerhalb einer *Store*-Methode in einen Stream geschrieben wurden, wieder zu laden. Siehe auch: *TGroup.Load*, *TView.SetPeerViewPtr*, *TGroup.Store*.

GetState

```
function GetState(AState: Word) : Boolean;
```

Über diese Funktion wird getestet, ob sich das *TView*-Objekt in einem bestimmten Zustand befindet; sie gibt True zurück, wenn die in *AState* gesetzten Bits denen in dem Feld *State* entsprechen. Siehe auch: *State*, *TView.SetState*.

GrowTo

```
procedure GrowTo(X, Y: Integer);
```

Vergrößert den Bereich des Objekts um die in *X* und *Y* übergebenen Werte.

Hide

```
procedure Hide;
```

Löscht das Bit *sfVisible* in *State*, wodurch das Objekt vom Bildschirm entfernt wird. Siehe auch: *sfVisible*.

HideCursor

```
procedure HideCursor;
```

Löscht das Bit *sfCursorVis* in *State*, was bewirkt, daß der Cursor nicht mehr ausgegeben wird. Siehe auch: *sfCursorVis*.

KeyEvent

```
procedure KeyEvent(var Event: TEvent);
```

Wartet auf das nächste verfügbare Tastatur-Ereignis; andere Ereignisse werden ignoriert. Sobald verfügbar, wird das Tastaturereignis über *Event* zurückgegeben. Siehe auch: *TView.GetEvent*, *TView.EventAvail*.

Locate

```
procedure Locate(var Bounds: TRect);
```

Setzt die Begrenzungen des Objekts auf die in *Bounds* übergebenen Werte. Das *View*-Objekt wird an der neuen Position ausgegeben. Diese Methode überprüft zunächst mittels *SizeLimits*, ob die in *Bounds* angegebenen Werte gültig sind, und ruft dann *ChangeLimits* auf. Siehe auch: *Size*, *Origin*, *TView.CalcBounds*, *TView.GrowTo*, *TView.MoveTo*.

MakeFirst

```
procedure MakeFirst;
```

Bewegt das diese Methode aufrufende Objekt an den Anfang der zu einer Gruppe gehörenden *TView*-Objekte und hat diesselbe Wirkung wie *PutInFrontOf*(*Owner^, First*). Je nach der Lage dieser Objekte und abhängig davon, ob diese Objekte überhaupt sichtbar sind, kann es sein, daß sie ganz oder teilweise überdeckt werden. Siehe auch: *TView.PutInFrontOf*.

MakeGlobal

```
procedure MakeGlobal(Source: TPoint; var Dest: TPoint);
```

Konvertiert die relativ zum Bereich des *TView*-Objekts definierten Koordinaten in *Source* in relativ zum Bildschirm festgelegte Werte und gibt diese Werte in *Dest* zurück. Siehe auch: *TView.MakeLocal*.

MakeLocal

```
procedure MakeLocal(Source: TPoint; var Dest: TPoint);
```

Konvertiert die relativ zum Bildschirm definierten Koordinaten in *Source* in relativ zum Bereich des *TView*-Objekts festgelegte Werte und gibt diese Werte in *Dest* zurück. Über diese Methode wird ermittelt, in welchem *TView*-Objekt ein Mausereignis stattfand, zum Beispiel: *MyWindow.MakeLocal*(*Event.Where, MouseLoc*);. Siehe auch: *TView.MakeGlobal*, *TView.MouseInView*.

MouseEvent

```
function MouseEvent(var Event: TEvent; Mask: Word):
                                        Boolean;
```

Gibt True zurück, wenn das nächste verfügbare Mausereignis eines der in *Mask* angegebenen Ereignisse, und False, wenn dieses Mausereignis das Ereignis *evMouseUp* ist. Das Ereignis selbst ist dann über *Event* verfügbar. Diese Methode erlaubt Ihnen, den Maus-Operationen zu folgen, während ein Mausknopf niedergehalten wird, wenn beispielsweise der Benutzer mittels der Maus in einem Editor einen Block markiert.

Der folgende Programmausschnitt zeigt eine entsprechende *HandleEvent*-Methode:

```
procedure TMyView.HandleEvent(var Event: TEvent);
begin
  TView.HandleEvent(Event);
  case Event.What of
    evMouseDown:
      begin
        repeat
          MakeLocal(Event.Where, Mouse);
          SetCursor(Mouse.X, Mouse.Y);
        until not MouseEvent(Event, evMouseMove);
      ClearEvent(Event);
    end;
    ...
  end;
end;
```

Siehe auch: *TView.KeyEvent*, *TView.GetEvent*.

MouseInView `function MouseInView(Mouse: TPoint): Boolean;`

Gibt True zurück, wenn die in *Mouse* übergebenen Koordinaten (die relativ zum Bildschirm zu verstehen sind) sich innerhalb des Bereichs des *View*-Objekts befinden. Siehe auch: *TView.MakeLocal*.

MoveTo `procedure MoveTo(X, Y: Integer);`

Setzt das Feld *Origin* auf die Werte *X* und *Y*, die relativ zum Bereich der Gruppe, der das Objekt angehört, anzugeben sind. Das Feld *Size* ändert sich nicht. Siehe auch: *Origin*, *Size*, *TView.Locate*.

NextView `function NextView: PView;`

Gibt einen Zeiger auf die nächste zur Gruppe gehörende Subview zurück oder **nil**, wenn das diese Methode aufrufende Objekt die letzte *Subview* der Gruppe ist. Siehe auch: *TView.Prev*, *TView.PrevView*.

NormalCursor `procedure NormalCursor;`

Löscht das Bit *sfCursorIns* in *State* und verändert die Form des Cursors in einen Unterstrich. Der Cursor selbst wird nur

dann sichtbar, wenn das Bit *sfCursorVis* gesetzt ist. Siehe auch: *sfCursorIns*, *sfCursorVis*, *TView.ResetCursor*, *TView.BlockCursor*, *TView.HideCursor*.

Prev

```
function Prev: PView;
```

Gibt einen Zeiger auf die dem aufrufenden Objekt vorangehende Subview der Gruppe zurück. Ist das aufrufende Objekt selbst die erste Subview der Gruppe, wird ein Zeiger auf die letzte der Gruppe angehörende Subview zurückgegeben. Sie sehen, daß *Prev* die Liste der Subviews einer Gruppe als zirkulär verkettet ansieht, wogegen die Methode *PrevView* sie als lineare Liste behandelt. Siehe auch: *TView.NextView*, *TView.PrevView*.

PrevView

```
function PrevView: PView;
```

Gibt einen Zeiger auf die dem aufrufenden Objekt vorangehende Subview der Gruppe zurück. Ist das aufrufende Objekt selbst die erste Subview der Gruppe, wird **nil** zurückgegeben. Sie sehen, daß *PrevView* die Liste der Subviews einer Gruppe als lineare Liste behandelt, wogegen die Methode *Prev* sie als zirkulär verkettet ansieht. Siehe auch: *TView.NextView*, *TView.Prev*.

PutEvent

```
procedure PutEvent(var Event: TEvent); virtual;
```

Bewirkt, daß das nächste Ereignis, das *GetEvent* ermittelt, das in *Event* gespeicherte Ereignis ist. Diese Methode wird oft dazu verwendet, Befehlsereignisse, etwa für die Kommunikation zwischen *TView*-Objekten, zu generieren, zum Beispiel:

```
Event.What := evCommand;
Event.Command := cmSaveAll;
Event.InfoPtr := nil;
PutEvent(Event);
```

TView.PutEvent ruft die *PutEvent*-Methode der Gruppe auf, zu der das Objekt gehört. Siehe auch: *TView.EventAvail*, *TView.GetEvent*, *TView.HandleEvent*.

PutInFrontOf

```
procedure PutInFrontOf(Target: PView);
```

Versetzt das Objekt, das diese Methode aufruft, vor die durch *Target* spezifizierte Subview einer Gruppe. Die Anweisung TView.PutInFrontOf(Owner^.First); hat die gleiche Wirkung wie ein Aufruf von *MakeFirst*. Je nach der Position anderer Objekte der Gruppe und abhängig davon, ob diese Objekte überhaupt sichtbar sind, kann es sein, daß sie ganz oder teilweise überdeckt werden. Siehe auch: *TView.MakeFirst*.

PutPeerViewPtr

```
procedure PutPeerViewPtr(var S: TStream; P: PView);
```

Diese Methode ist das Gegenstück zu *GetPeerViewPtr*, wird

von der Methode *TGroup.Store* aufgerufen und übergibt den Zeiger *P* an den *TStream*. Siehe auch: *TView.GetPeerViewPtr*, *TGroup.Load*, *TGroup.Store*.

Select

```
procedure Select;
```

Setzt das Bit *sfSelected* im *State*-Feld des aufrufenden Objekts und löscht dieses Bit im *State*-Feld des vorher selektierten Objekts. Dadurch wird das aufrufende *View*-Objekt sowohl selektiert als auch fokussiert. Ist im Feld *Options* des Objekts das Bit *ofTopSelect* gesetzt, wird es vor allen anderen zu der Gruppe gehörenden Objekten auf dem Bildschirm neu ausgegeben. Siehe auch: *sfSelected*, *sfFocused*, *ofTopSelect*, *TView.MakeFirst*.

SetBounds

```
procedure SetBounds(var Bounds: TRect);
```

Setzt den Bereich des *TView*-Objekts auf die in *Bounds* angegebenen Werte. Dem Feld *Origin* wird *Bounds.A*, dem Feld *Size* die Differenz aus *Bounds.B - Bounds.A* zugewiesen. Diese Methode wird innerhalb der *ChangeBounds*-Methode aufgerufen. Sie dürfen Sie nie direkt aufrufen. Siehe auch: *Origin*, *Size*, *TView.CalcBounds*, *TView.ChangeBounds*, *TView.GetBounds*, *TView.GetExtent*.

SetCommands

```
procedure SetCommands(Commands: TCommandSet);
```

Ändert die verfügbaren Befehle auf die in *Commands* angegebenen. Siehe auch: *TView.EnableCommands*, *TView.DisableCommands*.

SetCursor

```
procedure SetCursor(X, Y: Integer);
```

Setzt den Cursor auf die durch *X* und *Y* angegebene Position. Diese beiden Werte sind auf den Ursprung des Objekts zu beziehen. Beachten Sie, daß dieser Ursprung in der oberen linken Ecke auf den Koordinaten 0, 0 liegt. Siehe auch: *sfCursorVis*, *TView.MakeLocal*.

SetData

```
procedure SetData(var Rec); virtual;
```

Kopiert aus *Rec* soviele Bytes Daten, wie der Aufruf von *DataSize* zurückgibt. Siehe auch: *TView.GetData*, *TView.DataSize*.

SetState

```
procedure SetState(AState: Word; Enable: Boolean);
                   virtual;
```

Wenn *Enable* True ist, werden die in *AState* gesetzten Bits auch in *State* gesetzt; ist *Enable* False, werden die in *AState* gesetzten Bits in *State* gelöscht. *SetState* löst auch die den Statusänderungen entsprechenden Aktionen aus, einschließlich der Darstellung auf dem Bildschirm, wenn *sfVisible* geändert wurde.

SetState wird manchmal überschrieben, um abhängig von *State*, weitere Aktionen auzulösen. Zum Beispiel überschreibt *TFrame* diese Methode, um sich selbst neu darzustellen, wenn das Fenster selektiert oder über den Bildschirm bewegt wird:

```
procedure TFrame.SetState(AState: Word; Enable:
                          Boolean);

begin
  TView.SetState(AState, Enable);
  if AState and (sfActive + sfDragging) <> 0 then DrawView;
end;
```

Ein anderer Grund, diese Methode zu überschreiben, besteht darin, Befehle, auf die ein Objekt reagiert, zu deaktivieren oder wieder zu aktivieren:

```
procedure TMyView.SetState(AState: Word; Enable:
                           Boolean);
const
  MyCommands = [cmCut, cmCopy, cmPaste, cmClear];
begin
  TView.SetState(AState, Enable);
  if AState = sfSelected then
    if Enable then
      EnableCommands(MyCommands) else
      DisableCommands(MyCommands);
end;
```

Siehe auch: *State*, *TView.GetState*, *sfXXXX*-Konstanten.

Show

```
procedure Show;
```

Veranlaßt über einen Aufruf von *SetState* die Darstellung des Objekts auf dem Bildschirm. Siehe auch: *TView.SetState*.

ShowCursor

```
procedure ShowCursor;
```

Setzt das Bit *sfCursorVis* und veranlaßt dadurch, daß der Cursor sichtbar ist. Beachten Sie, daß per Voreinstellung der Cursor nicht ausgegeben wird. Siehe auch: *sfCursorVis*, *TView.HideCursor*.

SizeLimits

```
procedure SizeLimits(var Min, Max: TPoint); virtual;
```

Gibt über *Min* den kleinsten und über *Max* den größten Wert zurück, den das Feld *Size* annehmen kann. Sofern nicht überschrieben, gibt diese Methode den Wert *MinWinSize* für *Min* und den Wert des *Size*-Feldes der Gruppe für *Max* zurück. Überschreiben Sie diese Methode, um den Bereich, innerhalb

dessen das Objekt seine Größe ändern kann, zu begrenzen. Siehe auch: *MinWinSize*, *Size*.

Store

```
procedure Store(var S: TStream);
```

Übergibt das *TView*-Objekt an den *TStream*. Siehe auch: *TView.Load*.

TopView

```
function TopView: PView;
```

Gibt einen Zeiger auf das aktuelle modale *TView*-Objekt zurück.

Valid

```
function Valid(Command: Word: Boolean; virtual;
```

Diese Methode wird dazu benutzt, die Gültigkeit eines gerade über einen Konstruktor initialisierten Objekts oder die Gültigkeit des Objekts, das die Eigenschaft der Modalität verliert, zu überprüfen.

Hat der Parameter *Command* den Wert *cmValid* (=0), wird die Gültigkeit des Objekts nach seiner Initialisierung überprüft. Ist das Objekt erfolgreich initialisiert worden, muß *Valid* True zurückgeben. Ein anderer Wert von *Command* bedeutet, daß dieses Objekt (zum Beispiel ein Dialogfenster) die Modalität abgeben soll, wobei *Command* als Resultat zurückgegeben wird. Auch in diesem Fall ist die Gültigkeit des Objekts zu überprüfen.

Es liegt in der Verantwortung von *Valid*, den Benutzer zu informieren, wenn die Ungültigkeit des Objekts ermittelt wird. Dafür kann zum Beispiel die Routine *MessageBox* aus der Unit *StdDlg* verwendet werden. In dieser Unit finden Sie auch zahlreiche Objekttypen, die die Methode *Valid* überschreiben.

Wenn nicht überschrieben, gibt *TView.Valid* einfach True zurück. Siehe auch: *TGroup.Valid*, *TDialog.Valid*, *TProgram.ValidView*.

WriteBuf

```
procedure WriteBuf(X, Y, W, H: Integer; var Buf );
```

Schreibt den Inhalt des Puffers *Buf*, beginnend an der durch *X* und *Y* festgelegten Position auf den Bildschirm, wobei ein Bereich der Breite *W* und der Höhe *H* ausgefüllt wird. Diese Routine sollte nur innerhalb einer *Draw*-Methode aufgerufen werden. Der Parameter *Buf* ist normalerweise vom Typ *TDrawBuffer*, kann aber ein beliebiges **Array of** Word sein, wobei jedes Word das auszugebende Zeichen im niederwertigen und das verwendete Attribut im höherwertigen Byte speichern muß. Siehe auch: *TView.Draw*.

WriteChar

```
procedure WriteChar(X, Y: Integer; Ch: Char; Color:
                    Byte; Count: Integer);
```

Schreibt, beginnend an der durch *X* und *Y* bezeichneten Position, das Zeichen *Ch* *Count*-mal auf den Bildschirm. Das verwendete Attribut wird über den Eintrag *Color* in der aktuellen Farbpalette ermittelt. Diese Routine sollte nur innerhalb einer *Draw*-Methode aufgerufen werden. Siehe auch: *TView.Draw*.

WriteLine

```
procedure WriteLine(X, Y, W, H: Integer; var Buf );
```

Schreibt die im Puffer *Buf* gespeicherte Textzeile an der durch *X* und *Y* festgelegten Position auf den Bildschirm, und zwar in einen Bereich der Breite *W* und der Höhe *H*. Ist *H* größer 1, wird die Ausgabe *H*-mal wiederholt. Diese Routine sollte nur innerhalb einer *Draw*-Methode aufgerufen werden. Der Parameter *Buf* ist normalerweise vom Typ *TDrawBuffer*, kann aber ein beliebiges **Array of** Word sein, wobei jedes Word das auszugebende Zeichen im niederwertigen und das verwendete Attribut im höherwertigen Byte speichern muß. Siehe auch: *TView.Draw*.

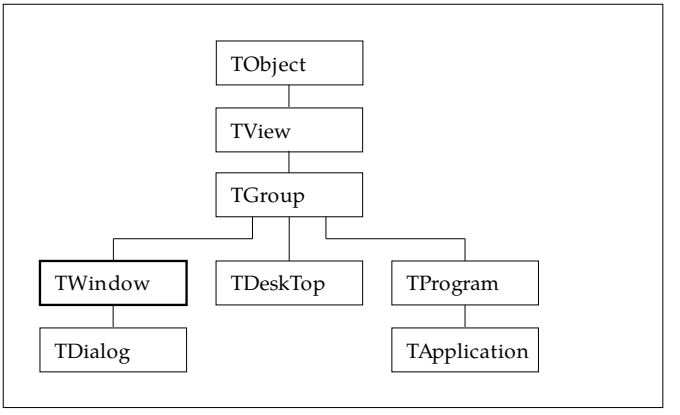
WriteStr

```
procedure WriteStr(X, Y: Integer; Str: String; Color:
                   Byte);
```

Schreibt an der durch *X* und *Y* bezeichneten Position den String *Str* auf den Bildschirm. Das verwendete Attribut wird über den Eintrag *Color* in der aktuellen Farbpalette ermittelt. Diese Routine sollte nur innerhalb einer *Draw*-Methode aufgerufen werden. Siehe auch: *TView.Draw*.

TWindow

VIEWS



TWindow ist ein spezialisierter Nachkomme von *TGroup* und damit selbst eine Gruppe, zu der üblicherweise ein *TFrame*-Objekt gehört, das ein *TScroller*-Objekt umschließt, und da-

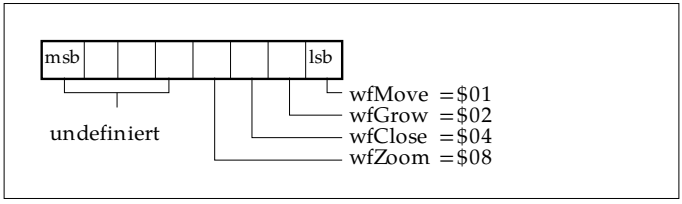
rüber hinaus auch ein oder zwei *TScrollBar*-Objekte umfaßt. Diese Objekte machen ein *TWindow*-Objekt auf dem Bildschirm sichtbar; *TFrame* versieht das Fenster mit einen Rahmen, der Platz bietet für einen Titel und verschiedene Icons enthält, die es ermöglichen, mittels der Maus die Position des Fensters und seine Größe zu ändern. Zu den Fähigkeiten eines *TWindow*-Objekts zählt auch, mit Rollbalken umgehen zu können. Hat das Feld *Number* einen Wert zwischen 1 und 9, kann das Fenster über die Tastenkombination [Alt][Ziffer] selektiert werden.

Felder

Flags

```
Flags: Byte;
```

Dieses Feld legt einige Eigenschaften des Fensters fest; die einzelnen Bits haben folgende Bedeutung:



Detaillierter sind diese Optionen unter dem Stichwort »*wfXXXX-Konstanten*« in Kapitel 15, »Außerhalb der Objekte« besprochen.

ZoomRect

```
ZoomRect: TRect;
```

Speichert die Begrenzungen des nicht gezoomten Fensters.

Number

```
Number: Integer;
```

Die Nummer des Fensters. Enthält dieses Feld einen Wert zwischen 1 und 9, erscheint diese Nummer oben rechts im Rahmen; das Fenster kann dann über die Tastenkombination [Alt][Ziffer] selektiert werden.

Palette

```
Palette: Integer;
```

Gibt an, welche Farbpalette für dieses Fenster verwendet wird, und kann die Werte *wpBlueWindow*, *wpCyanWindow* und *wpGrayWindow* annehmen. Voreingestellt ist die Palette *wpBlueWindow*. Siehe auch: *TWindow.GetPalette*, *wpXXXX*-Konstanten.

Frame

```
Frame: PFrame;
```

Zeigt auf das *TFrame*-Objekt, das zu dieser Gruppe gehört.

Title

```
Title: PString;
```

Zeigt auf den String, der als (optionaler) Titel des Fensters erscheint.

Methoden

Init

```
constructor Init(var Bounds: TRect; ATitle: TTitleStr;
                 ANumber: Integer);
```

Initialisiert über *TGroup.Init*(*Bounds*) ein Fenster in der in *Bounds* angegebenen Größe. Diese Werte werden auch im Feld *ZoomRect* gespeichert. In *Options* werden die Bits *ofSelectable* und *ofTopSelect*, in *Flags* die Bits *wfMove*, *wfGrow*, *wfClose* und *wfZoom* gesetzt. Den Feldern *Title* und *Number* werden *ATitle* und *ANumber* zugewiesen. Schließlich ruft dieser Konstruktor *TFrame.Init* auf und fügt dieses Objekt der Gruppe hinzu. Siehe auch: *TGroup.Init*, *TFrame.Init*.

Load

```
constructor Load(var S: TStream);
```

Lädt ein *TWindow*-Objekt aus dem angegebenen Stream. Siehe auch: *TGroup.Load*, *TWindow.Store*.

Done

```
destructor Done; virtual;
```

Gibt die zur Gruppe *TWindow* gehörenden Objekte, dann diese selbst frei.

Close

```
procedure Close; virtual;
```

Ruft die Methode *Valid* mit *cmClose* als *Command*-Parameter auf und schließt, wenn *Valid* True zurückgibt, über den Destruktor *Done* das Fenster. Siehe auch: *TWindow.Valid*, *TWindow.Done*.

GetPalette

```
function GetPalette: PPalette; virtual;
```

Diese Methode gibt einen Zeiger auf die aktuell verwendete Farbpalette zurück.

GetTitle

```
function GetTitle(MaxSize: Integer): TTitleStr;
                  virtual;
```

Enthält das Feld *Title* den Wert **nil**, gibt diese Methode einen leeren String, andernfalls *Title*^ zurück. Ist dieser String länger als MaxSize Zeichen, sollte diese Methode versuchen, ihn zu verkürzen, denn sonst wird nach MaxSize Zeichen abgeschnitten.

HandleEvent

```
procedure HandleEvent(var Event: TEvent); virtual;
```

Ruft zunächst *TGroup.HandleEvent* auf und arbeitet dann alle Ereignisse ab, die das Fenster selbst betreffen, sofern dieses selektiert ist. Das schließt Mausereignisse ein, die das Fenster schließen, es über den Bildschirm bewegen oder seine Größe

ändern, wenn *Flags* das zuläßt. Dabei werden die in *DragMode* definierten Grenzen berücksichtigt. Zuständig ist diese Methode auch für Tastaturereignisse wie [Tab] oder [Shift][Tab], die das folgende bzw. vorhergehende Fenster selektieren, sowie [Alt][Nummer], die ein Fenster über seine Nummer selektieren. Siehe auch: *TGroup.HandleEvent*.

InitFrame

```
procedure InitFrame; virtual;
```

Erstellt einen neuen Rahmen für das Fenster. Diese Methode legt in dem Feld *Frame* einen Zeiger auf diesen Rahmen ab, stellt diesen aber nicht dar. Sie müssen *TFrame.Draw* aufrufen, damit der Rahmen ausgegeben wird. Siehe auch: *TFrame.Draw*.

SetState

```
procedure SetState(AState: Word; Enable: Boolean);
                   virtual;
```

Ist *Enable* True, werden alle in *AState* gesetzten Bits auch in *State* gesetzt oder – wenn *Enable* False ist – gelöscht. Abhängig von *State* sind Befehle für das Fenster aktiviert oder deaktiviert. So können zum Beispiel – wenn das Bit *sfActive* in *State* gesetzt ist – die Befehle *cmNext*, *cmResize*, *cmClose* und *cmZoom* auf das Fenster angewendet werden, wenn *GrowMode* dies zuläßt. Siehe auch: *TGroup.SetState*, *EnableCommands*, *DisableCommands*.

SizeLimits

```
procedure SizeLimits(var Min, Max: TPoint); virtual;
```

Setzt die Minimal- und Maximalwerte, die das Fenster in seiner Größe nicht unter- bzw. überschreiten darf. Dazu ruft diese Methode *TView.SizeLimits*(*Max, Min*) auf und setzt dann *Min.X* auf *MinWinSize.X* und *Min.Y* auf *MinWinSize.Y*. Siehe auch: *TView.SizeLimits*.

StandardScrollBar

```
function StandardScrollBar(AOptions : Word):
                                PScrollBar;
```

Diese Methode initialisiert einen Rollbalken, fügt dieses Objekt in die Gruppe ein und gibt einen Zeiger darauf zurück. Die Größe dieses Rollbalkens ist so bemessen, daß er keine Ecke des Rahmens überdeckt, so daß es immer möglich ist, mittels der Maus die Fenstergröße zu ändern.

Der Parameter *AOptions* hat entweder den Wert *sbHorizontal*, wodurch ein horizontaler Rollbalken am unteren Fensterrand definiert wird, oder den Wert *sbVertical*, wodurch ein vertikaler Rollbalken am rechten Fensterrand definiert wird. Ist in *AOptions* auch das Bit *sbHandleKeyboard* gesetzt, reagiert der Rollbalken über Mausereignisse hinaus auch auf Tastaturereignisse. Siehe auch: *sbXXXX*-Konstanten.

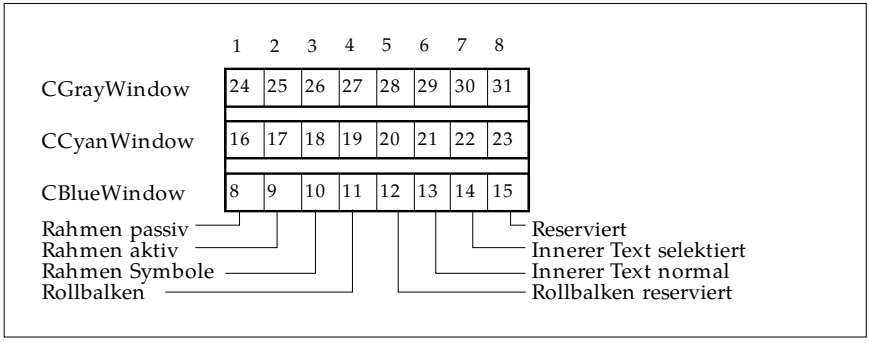
Store `procedure Store(var S: TStream);`

Schreibt über *TGroup.Store(S)* das Fenster und alle zu dieser Gruppe gehörenden Subviews in den angegebenen Stream. Dann werden auch die Felder *Flags*, *Frame* und *Title* mittels *S.Write* in dem Stream gespeichert. Siehe auch: *TGroup.Store*, *TWindow.Load*.

Zoom `procedure Zoom; virtual;`

Diese Methode zoomt das Fenster und wird normalerweise als Reaktion auf einen *cmZoom*-Befehl (ausgelöst durch Anklicken des Zoomfeldes) aufgerufen. Sie berücksichtigt automatisch die Größe des Fensters, seine Position innerhalb der Gruppe, der es angehört, und den Wert des Feldes *ZoomRect*. Siehe auch: *cmZoom*, *ZoomRect*.

Farbpalette Benutzt wird für Textfenster die Palette *CBlueWindow*, für Fenstern zur Ausgabe von Meldungen die Palette *CCyanWindow* und für Dialogfenster die Palette *CGrayWindow*.



15. Außerhalb der Objekte

Dieses Kapitel beschreibt in alphabetischer Ordnung alle Elemente in Turbo Vision, die keine Objekte sind. Ein Verzeichnis der Standardobjekte finden Sie in Kapitel 14, »Verzeichnis der Objekte«.

Hier finden Sie eine Aufstellung der Datentypen, Konstanten, Variablen, Prozeduren und Funktionen, die in den zu Turbo Vision gehörenden Units definiert sind. Die Einträge in diesem Verzeichnis entsprechen in etwa dem folgenden Beispiel:

Beispiel-Prozedur

UNIT-NAME

Deklaration	`procedure Beispiel(AParameter);`
Funktion	Beschreibung der Funktion; Hinweise zur Verwendung und zu den Funktionsparametern.
Querverweis	auf andere Einträge, die mit diesem in Zusammenhang stehen.

Abstract-Prozedur

OBJECTS

Deklaration	`procedure Abstract;`
Funktion	Diese Prozedur löst den Laufzeitfehler 211 aus, wenn eine abstrakte Methode aufgerufen wird. Wenn Sie einen *abstrakten Objekttyp* definieren, dann rufen Sie diese Prozedur in all jenen virtuellen Methoden auf, die in den von diesem Objekt abgeleiteten Typen überschrieben werden müssen. Damit stellen Sie sicher, daß jeder Versuch, Instanzen des abstrakten Objekttyps zu verwenden, fehlschlagen wird.
Querverweis	»Abstrakte Methoden« in Kapitel 4, »Die Hierarchie der Objekte«

Application-Variable APP

Deklaration
```
Application: PApplication = nil;
```

Funktion Dieser Zeiger auf ein Objekt des Typs *TApplication* wird von *TProgram.Init* (aufgerufen von *TApplication.Init*) auf *@Self* und vor der Beendigung des Programms von *TProgram.Done* (aufgerufen von *TApplication.Done*) auf **nil** gesetzt. Während ein Turbo-Vision-Programm läuft, zeigt also *Application* auf die von diesem Programm definierte Instanz von *TApplication*.

Querverweis *TProgram.Init*, *TApplication.Init*

AppPalette-Variable APP

Deklaration
```
AppPalette: Integer = apColor;
```

Funktion Wählt eine der drei verfügbaren Farbpaletten für ein Programm aus (*apColor*, *apBlackWhite* oder *apMonochrome*). *AppPalette* wird von *TProgram.InitScreen* mit dem Wert belegt, der dem ermittelten Bildschirmadapter entspricht, und *TProgram.GetPalette* gibt die passende Farbpalette zurück. Wollen Sie eine andere Farbpalette wählen, müssen Sie *TProgram.InitScreen* überschreiben.

Querverweis *TProgram.InitSreen*, *apXXXX*-Konstanten

apXXXX-Konstanten APP

Funktion Konstanten, deren Namen mit dem dem Präfix »*ap*« beginnen, bezeichnen die Farbpaletten, die einem Turbo-Vision-Programm zur Verfügung stehen.

Werte Turbo Vision definiert die folgenden Paletten-Konstanten:

Konstante	Wert	Bedeutung
apColor	0	Palette für Farbbildschirme
apBlackWhite	1	Palette für LCD-Bildschirme
apMonochrome	2	Palette für Monochrom-Bildschirme

Querverweis *AppPalette*, *TProgram.GetPalette*

AssignDevice-Prozedur TEXTVIEW

Deklaration
```
procedure AssignDevice(var T: Text;
                       screen: PTextDevice);
```

Funktion Ordnet die Textdatei *T* dem durch *Screen* referenzierten Objekt des Typs *TTextDevice* zu. Diese Prozedur arbeitet genau wie die Standardprozedur *Assign*, mit der Ausnahme, daß kein Dateiname anzugeben ist. Stattdessen verbindet *AssignDevice* die Datei-Variable mit *Screen*, wobei *Screen* in den ersten vier Bytes des Feldes *TextRec.UserData* abgelegt wird. Ein- und Ausgabeoperationen über die Datei *T* beziehen sich nach Aufruf dieser Prozedur auf das über *Screen* referenzierte *TTextDevice*-Objekt. Dabei werden dessen virtuelle Methoden *StrRead* und *StrWrite* benutzt. Da *TTextDevice* ein abstrakter Objekttyp ist, zeigt *Screen* in der Regel auf eine Instanz von *TTerminal*, ein Objekt, das fernschreiber-ähnliche Funktionen bereitstellt.

Querverweis *TTextDevice*, *Dos.TextRec*, *TTerminal*

bfXXXX-Konstanten DIALOGS

Funktion Diese Werte können dem Feld *Flags* eines *TButton*- oder davon abgeleiteten Objekts zugeordnet werden. Ist in diesem Feld das Bit *bfNormal* gesetzt, erscheint der Aktionsschalter in den normalen Farben, d.h. er ist nicht der voreingestellte Schalter. Ist dagegen das Bit *bfDefault* gesetzt, wird zur Darstellung die Hervorhebungsfarbe verwendet und der Schalter kann sofort mit `[Enter]`»gedrückt« werden. In der Verantwortung des Programmierers liegt es, daß in einer Gruppe immer nur ein auf diese Weise voreingestellter Schalter definiert ist. Das Bit *bfLeftJust* bestimmt die Position der Beschriftung des Schalters. Ist es gelöscht, wird diese Beschriftung zentriert, ansonsten linksbündig ausgegeben.

Werte

Konstante	Wert	Bedeutung
bfNormal	$00	normaler Schalter
bfDefault	$01	voreingestellter aktiver Schalter
bfLeftJust	$02	Beschriftung linksbündig ausgeben
bfBroadcast	$04	Botschaften über *Message* weitergeben

Bemerkung: Das Bit *bfBroadcast* legt fest, ob das *TButton*Objekt ein Ereignis über die Methode *PutEvent* oder die Funktion *Message* generiert. Ist dieses Bit gelöscht, gibt der Schalter,

wenn aktiviert, über *PutEvent* ein Ereignis des Typs *evCommand* weiter:

```
Event.What := evCommand;
Event.Command := Command;
Event.InfoPtr := @Self;
PutEvent(Event);
```

Ist dieses Bit gesetzt, sendet der Schalter, wenn aktiviert, über *Message* ein Ereignis des Typs *evBroadcast* an die Gruppe, zu der er gehört:

```
Message(Owner, evBroadcast, Command, @Self);
```

Querverweis *TButton.Flags, TButton.MakeDefault, TButton.Draw*

ButtonCount-Variable DRIVERS

Deklaration `ButtonCount: Byte = 0;`

Funktion Speichert die Zahl der Knöpfe der Maus oder ist 0, wenn keine Maus installiert ist. Der Wert wird von der Prozedur *InitMouse* ermittelt und sollte nicht geändert werden. Sie können über diese Variable ermitteln, ob eine Maus an das System angeschlossen ist.

Querverweis *InitMouse*

CheckSnow-Variable DRIVERS

Deklaration `CheckSnow: Boolean;`

Funktion *CheckSnow* entspricht völlig der in der Standard-Unit *Crt* definierten gleichnamigen Variablen. Ist sie *True*, finden Schreibaktionen in den Bildschirmspeicher nur während des horizontalen Strahlrücklaufs statt. Dadurch wird »Schneegestöber«, das ältere CGA-Adapter auf den Bildschirm bringen, vermieden.
CheckSnow wird von *InitVideo* auf True gesetzt, wenn ein CGA-Adapter ermittelt wurde. Nach dem Aufruf von *InitVideo* können Sie diese Variable jederzeit auf False setzen, um die Bildschirmausgabe zu beschleunigen.

Querverweis *InitVideo, Crt.CheckSnow*

ClearHistory-Prozedur HISTLIST

Deklaration
```
procedure ClearHistory;
```
Funktion Entfernt alle Strings aus allen Eingabeaufzeichnungslisten.

ClearScreen-Prozedur DRIVERS

Deklaration
```
procedure ClearScreen
```
Funktion Diese Prozedur löscht den Bildschirm, gleichgültig welcher der Modi, die Turbo Vision unterstützt, gerade aktiv ist. *ClearScreen* geht davon aus, daß zuvor *InitVideo* aufgerufen wurde. Die Methode *TApplication.Run* zum Beispiel tut dies automatisch.

Querverweis *InitVideo*

cmXXXX-Konstanten VIEWS

Funktion Konstanten, deren Namen mit dem Präfix »*cm*« beginnen, stehen für die vordefinierten Befehle. In Turbo Vision wird ein Befehlsereignis in einem Record des Typs *TEvent* gespeichert, dessen Feld *Command* den Befehl selbst angibt. Da dieses Feld vom Typ Word ist, sind maximal 65535 Befehle möglich. Davon reserviert Turbo Vision für eigene Zwecke die Bereiche von 0 bis 99 und von 256 bis 999, so daß die Bereiche 100 bis 255 und 1000 bis 65535 für Befehle zur Verfügung stehen, die der Programmierer definiert.

Werte Die folgenden vordefinierten Befehle werden von den Standard-Objekten in Turbo Vision benutzt:

Konstante	Wert	Bedeutung
cmValid	0	Wird an *TView.Valid* übergeben, damit diese Methode die Gültigkeit eines gerade instantiierten *View*-Objekts überprüfen kann.
cmQuit	1	Veranlaßt *TProgram.HandleEvent* über *EndModal*(*cmQuit*) das Programm zu beenden. In der Regel stellt die Statuszeile oder ein Menüeintrag eine Option bereit, die diesen Befehl mit *kbAltX* verbindet.

Konstante	Wert	Bedeutung
cmError	2	Wird von keinem Objekt behandelt. Sie können diese Konstante für nicht implementierte oder nicht unterstützte Befehle benutzen.
cmMenu	3	Bewirkt, daß *TMenuView.HandleEvent* die Methode *ExecView* aufruft, wodurch Menüoptionen ausgewählt werden können. Ist eine Option gewählt, wird über *PutEvent* eine neuer Befehl abgesetzt. In der Regel stellt die Statuszeile eine Option bereit, die diesen Befehl mit *kbF10* verbindet.
cmClose	4	Wird von *TWindow.HandleEvent* abgearbeitet, wenn das Feld *InfoPtr* des Ereignisrecords den Wert **nil** speichert oder auf das Fensterobjekt zeigt. Ist dieses Objekt modal (zum Beispiel ein Dialogfenster), wird über *PutEvent* der Befehl *cmCancel* abgesetzt. Ist das Objekt nicht modal, wird die Methode *Close* dieses Fensterobjekts aufgerufen, sofern es geschlossen werden kann (siehe *wfClose*). Ein Anklicken des Schließfelds generiert einen TEvent-Record, dessen Feld *Command* mit dem Wert *cmClose* belegt ist und dessen Feld *InfoPtr* einen Zeiger auf das Fensterobjekt speichert. In der Regel stellt die Statuszeile oder ein Menüeintrag eine Option bereit, die diesen Befehl mit *kbAltF3* verbindet.
cmZoom	5	Bewirkt, daß *TWindow.HandleEvent* die Methode *TWindow.Zoom* aufruft, sofern das Fenster gezoomt werden darf (siehe *wfZoom*) und das Feld *InfoPtr* des Ereignisrecords **nil** oder einen Zeiger auf das Fenster speichert.

Konstante	Wert	Bedeutung
		Ein Anklicken des Schließfelds generiert einen *TEvent*-Reord,dessen Feld *Command* mit demWert *cmZoom* belegt ist und dessen Feld *InfoPtr* einen Zeiger auf das Fensterobjekt speichert. In der Regel stellt die Statuszeile oder ein Menüeintrag eine Option bereit, die diesen Befehl mit *kbF5* verbindet.
cmResize	6	Bewirkt, daß *TWindow.HandleEvent* die Methode *TView.DragView* aufruft, sofern die Größe des Fensters geändert werden darf (siehe *wfMove* und *wfGrow*). In der Regel stellt die Statuszeile oder ein Menüeintrag eine Option bereit, die diesen Befehl mit *kbCtrlF5* verbindet.
cmNext	7	Bewirkt, daß *TDeskTop.HandleEvent* das letzte zur Arbeitsfläche gehörende Fenster vor alle anderen Fenster bewegt. In der Regel stellt die Statuszeile oder ein Menüeintrag eine Option bereit, die diesen Befehl mit *kbF6* verbindet.
cmPrev	8	Bewirkt, daß *TDeskTop.HandleEvent* das erste zur Arbeitsfläche gehörende Fenster hinter allen andern Fenstern verbirgt. In der Regel stellt die Statuszeile oder ein Menüeintrag eine Option bereit, die diesen Befehl mit *kbShiftF6* verbindet.

Die folgenden Standardbefehle werden in Dialogfenstern verwendet:

Konstante	Wert	Bedeutung
cmOk	10	»Ok«-Aktionsschalter gedrückt
cmCancel	11	»Cancel«-Aktionsschalter gedrückt
cmYes	12	»Yes«-Aktionsschalter gedrückt

Konstante	Wert	Bedeutung
cmNo	13	»No«-Aktionsschalter gedrückt
cmDefault	14	Voreingestellter Aktionsschalter gedrückt

Die Befehle *cmOK*, *cmCancel*, *cmYes* und *cmNo* bewirken, daß *TDialog.HandleEvent* den Dialog über *EndModal* beendet und den Befehlswert zurückliefert. In der Regel enthält ein Dialogfenster zumindest einen Aktionsschalter, dem einer der oben aufgeführten Befehle zugeordnet ist. *TDialog.Handle-Event* generiert den Befehl *cmCancel* auch dann, wenn das Tastaturereignis »Esc-Taste gedrückt« eintrifft.
Der Befehl *cmDefault* führt dazu, daß *TButton.HandleEvent* den voreingestellten Schalter aktiviert (siehe *bfDefault*). *TDialog.HandleEvent* generiert den Befehl *cmDefault* auch dann, wenn das Tastaturereignis »Enter-Taste gedrückt« eintrifft.
Die folgenden Befehle werden von den in Turbo Vision definierten *View*-Objekten benutzt:

Konstante	Wert	Bedeutung
cmReceivedFocus	50	*TView.SetState* sendet über die
cmReleasedFocus	51	Funktion *Message* immer dann ein *evBroadcast*-Ereignis mit einem dieser Werte an *TView.Owner*, wenn *sfFocused* sich ändert. Das Feld *InfoPtr* des Ereignisrecords zeigt auf das aussendende *View*-Objekt. Dadurch teilt es allen anderen zu der Gruppe gehörenden Objekten mit, daß es den Fokus übernommen oder verloren hat, und fordert sie auf, ihren Status zu überprüfen und sich gegebenenfalls neu darzustellen. Ein *TLabel*-Objekt beispielsweise stellt sich in der Hervorhebungsfarbe dar, wenn das Objekt, das es beschriftet, den Fokus übernommen hat.
cmCommandSet-Changed	52	Wenn die Methode *TProgram.Idle* eine Änderung der aktuell verfügbaren Befehle ermittelt, generiert diese Methode ein *evBroadcast*-Ereignis mit diesem Wert (geändert werden kann der

Konstante	Wert	Bedeutung
		Befehlssatz über die in *TView* definierten Methoden *EnableCommands*, *DisableCommands* und *SetCommands*). Dieses Ereignis wird allen direkt oder indirekt zu *TApplication* gehörenden *View*-Objekten übermittelt, sofern deren *EventMask* die Wahrmehmung eines *evBroadCast*-Ereignisses zuläßt. Wenn die Darstellung eines Objekts von der Änderung des Befehlssatzes betroffen ist, muß es sich entsprechend neu auf dem Bildschirm ausgeben. Die Objekte *TButton*, *TMenuView* und *TStatusLine* zum Beispiel reagieren auf dem Empfang dieses Befehls, indem sie sich neu darstellen.
cmScrollBarChanged	53	Ein *TScrollBar*-Objekt benutzt die Funktion *Message*, um ein *evBroad-cast*-Ereignis mit einem dieser Werte auszusenden, wenn ein Feld dieses Objekts seinen Wert ändert oder der Rollbalken angeklickt wird. Das Feld *InfoPtr* des Ereignisrecords zeigt auf das *TScrollBar*-Objekt. Auf ein solches Ereignis reagieren die Objekte, mit denen dieser Rollbalken verbunden ist, zum Beispiel ein *TScroller*- oder ein *TListViewer*-Objekt.
cmScrollBar*Clicked*	54	
cmSelectWindowNum	55	Bewirkt, daß *TWindow.Handle* Event die Methode *TView.Select* aufruft, sofern der im Feld *InfoInt* des Ereignisrecords abgelegte Wert dem Wert von *TWindow.Number* entspricht. Drückt der Benutzer eine der Tastenkombinationen `[Alt][1]` bis `[Alt][9]`, sendet *TProgram.HandleEvent* ein *evBroadcast*-Ereignis ab; das Feld

Konstante	Wert	Bedeutung
		InfoInt des Ereignisrecords speichert einen Wert im Bereich 1 bis 9.
cmListItemSelected	56	Das Objekt *TListViewer* sendet über die Funktion Message ein Ereignis des Typs *evBroadcast* mit dem Wert *cmListItemSeleted* an die Gruppe, zu der es gehört, wenn ein Eintrag in der Liste selektiert wurde. Das kann entweder über einen Doppelklick erreicht werden oder indem der Leuchtbalken auf den Eintrag gesetzt und dann [Leertaste] gedrückt wird.
cmRecordHistory	60	Bewirkt, daß ein *THistory*-Objekt den Inhalt des Eingabefelds aufzeichnet, welches das mit ihm verbundene *TInputLine*-Objekt verwaltet. Wird ein Aktionsschalter »gedrückt«, sendet er diesen Befehl an die Gruppe, zu der er gehört. Dadurch bewirkt der Aktionsschalter, daß alle *THistory*-Objekte im gegebenen Dialogfenster Eingaben aufzeichnen.

Querverweis *TView.HandleEvent*, *TEvent*, *TCommandSet*

coXXXX-Konstanten

OBJECTS

Funktion Konstanten, deren Name mit »*co*« beginnen, dienen dazu, Fehler bei der Benutzung eines *TCollection*- oder davon abgeleiteten Objekts anzuzeigen.

Werte Die folgenden Fehler-Konstanten sind in Turbo Vision definiert:

Konstante	Wert	Bedeutung
coIndexError	-1	Index außerhalb des zulässigen Bereichs. Der Parameter *Info*, der der Methode *Error* übergeben wird, enthält den ungültigen Indexwert.

Konstante	Wert	Bedeutung
coOverflow	-2	Überlauf in der Kollektion. Der Methode *TCollection.SetLimit* gelang es nicht, die Kollektion auf die geforderte Größe zu erweitern. Der Parameter *Info*, der der Methode *Error* übergeben wird, enthält die angeforderte Größe.

Querverweis *TCollection*

CStrLen-Funktion DRIVERS

Deklaration `function CStrLen(S: String): Integer;`

Funktion Gibt die Länge des Strings *S* zurück, wobei eventuell darin enthaltene Tilde-Zeichen (~) nicht mitgerechnet werden. Wenn Sie dieser Funktion zum Beispiel den String *~B~roccoli* übergeben, erhalten Sie den Wert 8 zurück.

CtrlBreakHit-Variable DRIVERS

Deklaration `CtrlBreakHit: Boolean = False;`

Funktion Wird immer dann auf True gesetzt, wenn das Ereignis »Tastenkombination `[Ctrl][Break]` gedrückt« ermittelt wird. Wenn Ihr Programm auf dieses Ereignis reagieren soll, müssen Sie diese Variable abfragen. Sie können sie jederzeit auf False setzen.

Querverweis *SaveCtrlBreak*

CtrlToArrow-Funktion DRIVERS

Deklaration `function CtrlToArrow(KeyCode: Word): Word;`

Funktion Konvertiert den Tastencode *KeyCode*, den die Tastatur zurückliefert, wenn eine zu Wordstar kompatible Tastenkombination gedrückt wird, und gibt einen Wert zurück, der einer der *kbXXXX*-Konstanten entspricht. Ausgewertet wird nur das niederwertige Byte von *KeyCode*; die folgende Tabelle listet die Werte, die konvertiert werden. Andere Werte bewirken, daß *CtrlToArrow* den Wert *KeyCode* zurückgibt.

Taste	Lo(KeyCode)	Resultat
[Ctrl][A]	$01	kbHome
[Ctrl][D]	$04	kbRight
[Ctrl][E]	$05	kbUp
[Ctrl][F]	$06	kbEnd
[Ctrl][G]	$07	kbDel
[Ctrl][S]	$13	kbLeft
[Ctrl][V]	$16	kbIns
[Ctrl][X]	$18	kbDown

CursorLines-Variable **DRIVERS**

Deklaration
```
CursorLines: Word;
```

Funktion In dieser Variablen speichert *InitVideo* die vorgefundene Form des Cursors. *DoneVideo* setzt den Cursor vor der Beendigung des Programms wieder auf diesen Wert.

Querverweis *InitVideo*, *DoneVideo*, *TView.NormalCursor*, *TView.BlockCursor*, *TView.HideCursor*

DeskTop-Variable **APP**

Deklaration
```
DeskTop: PDeskTop = nil;
```

Funktion Diese Variable nutzen *TProgram* und *TApplication*, um einen Zeiger auf das *TDeskTop*-Objekt zu speichern. Der Gruppe *DeskTop^* werden in der Regel weitere *View*-Objekte eingefügt. Die Initialisierung der Variablen erfolgt durch *TProgramm.InitDeskTop*.

DisposeMenu-Prozedur **MENUS**

Deklaration
```
procedure DisposeMenu(Menu: PMenu);
```

Funktion Gibt den *PMenu* zugeordneten Speicherplatz frei (und auch den Speicherplatz, der von mit diesem Menü verbundenen Untermenüs belegt wird).

DisposeStr-Prozedur **VIEWS**

Deklaration
```
procedure DisposeStr(P: PString);
```

Funktion Gibt den Speicherplatz, den der über *P* referenzierte String belegt, wieder frei. Diese Prozedur darf nur auf Strings ange-

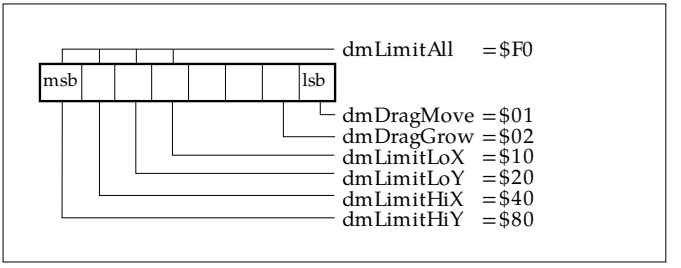
wendet werden, denen mittels der Funktion *NewStr* Heap-Speicher zugewiesen wurde.

Querverweis *NewStr*

dmXXXX-Konstanten VIEWS

Funktion Jedes *View*-Objekt besitzt das Feld *DragMode*, dessen Wert einerseits bestimmt, ob dieses Objekt seine Position oder Größe ändern kann. Ist dies der Fall, legt *DragMode* auch fest, wie das Objekt bewegt werden kann und ob es erlaubt ist, es aus dem Bereich der Gruppe, zu der es gehört, hinauszubewegen.

Werte Die Konstanten, deren Namen mit »*dm*« beginnen, dienen dazu, dem Feld *DragMode* die gewünschten Werte zuzuweisen,



und folgendermaßen definiert:

Konstante	Wert	Bedeutung
dmDragMove	$01	Ist dieses Bit gesetzt, können Sie das View-Objekt mit der Maus bewegen, wenn Sie den oberen Rahmen angeklickt haben.
dmDragGrow	$02	Ist dieses Bit gesetzt ist, kann das View-Objekt seine Größe ändern.
dmLimitLoX	$10	Ist dieses Bit gesetzt, kann die linke Seite des *View*-Objekts den Be-reich der Gruppe nicht verlassen.
dmLimitLoY	$20	Ist dieses Bit gesetzt, kann die obere Seite des *View*-Objekts den Bereich der Gruppe nicht verlassen.

Konstante	Wert	Bedeutung
dmLimitHiX	$40	Ist dieses Bit gesetzt, kann die rechte Seite des *View*-Objekts den Bereich der Gruppe nicht verlassen.
dmLimitHiY	$80	Ist dieses Bit gesetzt, kann die untere Seite des *View*-Objekts den Bereich der Gruppe nicht verlassen.
dmLimitAll	$F0	Ist dieses Bit gesetzt, kann das *View*-Objekt den Bereich der Gruppe nicht verlassen.

TView.Init setzt im Feld *DragMode* eines *View*-Objekts nur das Bit *dmLimitLoY*; alle anderen Bits sind gelöscht. Das *DragMode*-Feld wird bei einem *TWindow*-Objekt dazu verwendet, den Wert des an *DragView* zu übergebenden Parameters *Mode* festzulegen.

DoneEvents-Prozedur **DRIVERS**

Deklaration
```
procedure DoneEvents;
```

Funktion Beendet die *Event*-Verwaltung von Turbo Vision und muß vor dem Programmende aufgerufen werden. Die Methode *TApplication.Done* tut dies automatisch.

Querverweis *TApplication.Done, InitEvents*

DoneHistory-Prozedur **HISTLIST**

Deklaration
```
procedure DoneHistory;
```

Funktion Gibt den Speicherbereich, der über *InitHistory* belegt wurde, wieder frei. *TApplication.Done* ruft diese Prozedur automatisch auf.

Querverweis *InitHistory, HistoryBlock, TApplication.Done*

DoneMemory **MEMORY**

Deklaration
```
procedure DoneMemory;
```

Funktion Beendet die Speicherverwaltung von Turbo Vision, wobei alle über *GetBufMem* reservierten Bereiche freigegeben werden,

und muß vor dem Programmende aufgerufen werden. Die Methode *TApplication.Done* tut dies automatisch.

Querverweis *TApplication.Done, InitMemory*

DoneSysError-Prozedur DRIVERS

Deklaration `procedure DoneSysError;`

Funktion Restauriert die Interruptvektoren $09, $1B, $21, $23 und $24, die Turbo Vision übernahm, um kritische Fehler abzufangen. Diese Prozedur muß vor der Beendigung eines Turbo-Vision-Programms aufgerufen werden. Die Methode *TApplication.Done* tut dies automatisch.

Querverweis *TApplication.Done, InitSysError*

DoneVideo-Prozedur DRIVERS

Deklaration `procedure DoneVideo;`

Funktion Schaltet wieder auf den Videomodus um, den das Programm beim Start vorfand, und löscht den Bildschirm. Auch der Cursor wird wieder in der vorgefundenen Form ausgegeben. Diese Prozedur wird von *TApplication.Done* aufgerufen.

Querverweis *TApplication.Done, InitVideo, StartUp, StartUpMode*

DoubleDelay-Variable DRIVERS

Deklaration `DoubleDelay: Word = 8;`

Funktion Überschreitet der zeitliche Abstand zwischen zwei Mausklicks diesen Wert (anzugeben in 1/18,2 Bruchteilen einer Sekunde), werden diese nicht mehr als Doppelklick interpretiert. *GetMouseEvent* generiert immer dann ein *Double*-Ereignis, wenn der zeitliche Abstand zwischen zwei Mausklicks nicht größer ist als in dieser Variablen angegeben.

Querverweis *TEvent.Double, GetMouseEvent*

EmsCurHandle-Variable OBJECTS

Deklaration `EmsCurHandle: Word = $FFFF;`

Funktion Speichert den EMS-Handle, den ein *TEmsStream*-Objekt nutzt, um über die physikalische Page 0 auf den EMS-Speicher zuzu-

greifen. Damit eine zeitaufwendige Zuordnung der physikalischen Page vermieden wird, speichert *TEmsStream* den aktuellen Status der EMS-Speicherzuordnung ab. Nutzt Ihr Programm EMS-Speicher für andere Zwecke, müssen Sie *EmsCurHandle* und *EmsCurPage* auf den Wert $FFFF setzen. Dadurch ist *TEmsStream* gezwungen, die physikalische Seite neu zuzuordnen.

Querverweis *TEmsStream.Page*

EmsCurPage-Variable

OBJECTS

Deklaration
```
EmsCurPage: Word = $FFFF;
```

Funktion Ein *TEmsStream*-Objekt speichert in dieser Variablen die Nummer der aktuellen logischen Page im EMS-Speicher, die in der physikalischen Page 0 eingeblendet ist. Damit eine zeitaufwendige Zuordnung der physikalischen Page vermieden wird, speichert *TEmsStream* den aktuellen Status der EMS-Speicherzuordnung ab. Nutzt Ihr Programm EMS-Speicher für andere Zwecke, müssen Sie *EmsCurHandle* und *EmsCurPage* auf den Wert $FFFF setzen. Dadurch ist *TEmsStream* gezwungen, die physikalische Seite neu zuzuordnen.

Querverweis *TEmsStream.Handle*

ErrorAttr-Variable

VIEWS

Deklaration
```
ErrorAttr: Byte = $CF;
```

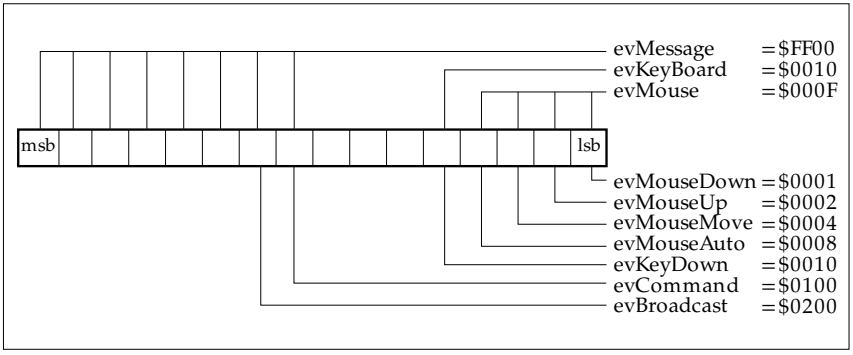
Funktion Dieses Byte wird als Bildschirmattribut zurückgegeben, wenn *TView.GetColor* das über einen Indexwert angesprochene Attribut aus der Farbpalette nicht ermitteln kann, weil dieser Indexwert außerhalb des zulässigen Bereichs liegt. Das Attribut *ErrorAttr* erscheint auf dem Bildschirm als leuchtendes, blinkendendes Weiß vor rotem Hintergrund.

Querverweis TView.GetColor

evXXXX-Konstanten

DRIVERS

Funktion Diese Konstanten werden zum einen im Feld *What* eines *TEvent*-Records zur Unterscheidung der verschiedenen Arten von Ereignissen benutzt. Zum anderen dienen sie dazu, dem Feld *EventMask* eines *View*-Objekts einen Wert zuzuweisen, der bestimmt, für welche Ereignisarten dieses Objekt zuständig ist.



Werte

Die in der folgenden Tabelle aufgelisteten Konstanten bezeichnen allgemeine Ereignis-Kategorien. Über sie können Sie überprüfen, ob ein Ereignis einer dieser Kategorien angehört, z.B. durch:

```
if Event.What and evMouse <> 0 then DoMouseEvent;
```

Konstante	Wert	Bedeutung
evNothing	$0000	Schon abgearbeitetes Ereignis
evMouse	$000F	Mausereignis
evKeyboard	$0010	Tastaturereignis
evMessage	$FF00	Ereignis der Kategorie Botschaft, Befehl oder Rundruf

Zwischen diesen allgemeinen Kategorien kann mittels der folgenden Bitmasken weiter differenziert werden:

Konstante	Wert	Bedeutung
evMouseDown	$0001	Maustaste gedrückt
evMouseUp	$0002	Maustaste losgelassen
evMouseMove	$0004	Maus wird bewegt
evMouseAuto	$0008	Maus bei gedrückter Taste bewegt
vKeyDown	$0010	Taste gedrückt
evCommand	$0100	Befehlsereignis
evBroadcast	$0200	Rundruf-Ereignis

Querverweis

TEvent, *TView.EventMask*, *GetKeyEvent*, *GetMouseEvent*, *Txxxx.Handle.Event*, *PositionalEvents*, *FocusedEvents*

FNameStr-Typ

OBJECTS

Deklaration
```
FNameStr: string[79];
```

Funktion Ein Stringtyp zur Aufnahme eines DOS-Dateinamens.

FocusedEvents-Variable

VIEWS

Deklaration
```
FocusedEvents: Word = evKeyBoard + evCommand;
```

Funktion Definiert die Ereignisklassen, die als *fokussierte Ereignisse* anzusehen sind. Über die Variablen *FocusedEvents* und *PositionalEvents* entscheidet *TGroup.HandleEvent*, in welcher Weise ein Ereignis an die zu dieser Gruppe gehörenden Objekte weitergegeben werden. Ist ein Ereignis weder den in *FocusedEvents* noch den in *PositionalEvents* definierten Ereignisklassen zuzuordnen, wird es als ein Ereignis des Typs *evBroadCast* behandelt.

Querverweis *PositionalEvents*, *TGroup.HandleEvent*, *TEvent*, *evXXXXKonstanten*

FormatStr-Prozedur

DRIVERS

Deklaration
```
procedure FormatStr(var Result: String;
                    Format: String; var Params);
```

Funktion Diese String-Formatierungsroutine arbeitet ähnlich wie die in C definierte Funktion **vsprintf.** Der zu übergebende String *Format* enthält Formatanweisungen, die diese Prozedur entsprechend der in *Params* enthaltenen Parametern auswertet. Der formatierte String wird dann über *Result* zurückgegeben. Der Parameter *Format* kann eine beliebige Anzahl von Formatanweisungen enthalten. Sie geben an, wie die in *Params* gegebenen Parameter zu formatieren sind. Die Anweisungen haben die Form %[-][nnn]X, wobei gilt:

% kennzeichnet den Beginn einer Formatanweisung.

[-] ist ein Minuszeichen, das nur dann anzugeben ist, wenn der Parameter linksbündig formatiert werden soll. Fehlt dieses Minuszeichen, wird der Parameter rechtsbündig formatiert.

[nnn] ist ebenfalls optional und gibt die Breite an, in der der Parameter auszugeben ist. Möglich sind Werte von 0 bis 255. 0 bedeutet, daß keine Breite angegeben ist; ein größerer Wert bewirkt, daß der Parameter in einem Feld von entsprechend vielen Zeichen ausgegeben wird.

X muß eines der folgenden Zeichen enthalten:

»s« bedeutet, daß der Parameter ein Zeiger auf einen String ist;

»d« bedeutet, daß der Parameter ein Longinteger-Wert ist, der als Dezimalwert zu formatieren ist;

»c« bedeutet, daß das niederwertigste Byte des Parameters als Zeichen anzusehen ist;

»x« bedeutet, daß der Parameter ein Longinteger-Wert ist, der als Hexadezimalwert zu formatieren ist.

Die folgende Tabelle zeigt einige Formatanweisungen, die davon ausgehen, daß der Parameter ein Zeiger auf den String »Versuch« ist, und die entsprechenden Ergebnisse:

Anweisung	Ergebnis
%8s	» Versuch«
%-8s	»Versuch «
%4s	»such«
%-3s	»Ver«
%08s	»0Versuch«
%-08s	»Versuch0«

Params ist ein untypisierter **var**-Parameter. Die Zahl der darin übergebenen Parameter muß der Zahl der Formatanweisungen im String *Format* entsprechen. *Params* kann als **Array of** LongInt oder **Array of** Pointer definiert werden; der Index dieses Arrays muß mit 0 beginnen. *Params* kann aber auch als Record, dessen Felder vom Typ LongInt oder Pointer sind, definiert werden.
Nehmen wir an, Sie wollten folgende Fehlermeldung ausgeben: Fehler in Datei [Dateiname] in Zeile [Zeilennummer]. Dazu müßten Sie in Format den String 'Fehler in Datei %s in Zeile %d' übergeben. *Params* muß einen Zeiger auf den String, der den Dateinamen speichert, und einen Longinteger-Wert, der der Nummer der Zeile entspricht, enthalten. Diese Werte können Sie entweder als Array oder als Record übergeben.
Der folgende Programmausschnitt zeigt die notwendigen (und möglichen) Typdeklarationen und Variablenzuweisungen:

```
type
  ErrMagRec = record
    FileName: PString;
    LineNo: LongInt;
  end;

ErrMsgArray = array[0..1] of LongInt;
```

```
const
  TemplateMsg = 'Fehler in Datei %s in Zeile %d';

var
  MyFileName: FNameStr;

  OopsRec: ErrMsgRec;
  DarnArray: ErrMsgArray;
  TestStr: String;

begin
  MyFileName := 'WARTHOG.ASM';

  with OopsRec do
  begin
    FileName := @MyFileName;
    LineNo := 42;
  end;
  FormatStr(TestStr, TemplateMsg, OopsRec);
  WriteLn(TestStr);

  DarnArray[0] := LongInt(@MyFileName);
  DarnArray[1] := 24;
  FormatStr(TestStr, TemplateMsg, DarnArray);
  WriteLn(TestStr);

end.
```

Querverweis *SystemError, TParamText*

FreeBufMem-Prozedur

MEMORY

Deklaration `procedure FreeBufMem(P: Pointer);`

Funktion Gibt den als Puffer genutzten Speicherbereich, auf den *P* zeigt, wieder frei.

Querverweis *GetBufMem, DoneMemory*

GetAltChar-Funktion

DRIVERS

Deklaration `function GetAltChar(KeyCode: Word): Char;`

Funktion Ermittelt, welche [Alt]-Tastenkombination den Wert von *KeyCode* liefert, und gibt das Zeichen zurück, das auf dieser Taste liegt.

Querverweis *GetAltCode*

GetAltCode-Funktion DRIVERS

Deklaration `function GetAltCode(Ch: Char): Word;`

Funktion Gibt den Wert zurück, den auch die Tastatur liefert, wenn eine Kombination aus `[Alt]`und dem Zeichen *Ch* gedrückt wird.

Querverweis *GetAltChar*

GetBufMem-Prozedur MEMORY

Deklaration `procedure GetBufMem(var P: Pointer; Size: Word);`

Funktion Reserviert einen Speicherbereich von *Size* Bytes Größe und gibt in *P* einen Zeiger darauf zurück. Ist kein Speicher für den Puffer frei, wird **nil** zurückgegeben. Im Unterschied zu Speicherreservierungen über *New*, *GetMem* oder *MemAlloc*, gibt die Speicherverwaltung von Turbo Vision die über diese Routine reservierten Puffer wieder frei, wenn dies notwendig ist, um normalen Speicheranforderungen nachkommen zu können. Der Zeiger *P* wird dann auf **nil** gesetzt. Über *FreeBufMem* können Sie einen von *GetBufMem* angelegten Puffer auch selbst freigeben. Diese Puffer werden im noch nicht reservierten Heapspeicher, also in dem Bereich zwischen *HeapPtr* und *HeapEnd* angelegt, und nutzen auch den von dem Programm definierten Sicherheitsbereich.
Turbo Vision nutzt diese Puffer dazu, die Bildschirmausgabe von *TGroup*-Objekten einschließlich ihrer Subviews zu beschleunigen, sofern in deren Feld *Options* das Bit *ofBuffered* gesetzt ist.

Querverweis *FreeBufMem*, *MemAlloc*, *InitMemory*, *TGroup.Draw*

GetKeyEvent-Prozedur DRIVERS

Deklaration `procedure GetKeyEvent(var Event: TEvent);`

Funktion Überprüft mittels des Tastaturinterrupts $16, ob ein Tastendruck anliegt. Ist das der Fall, wird das Feld *What* des übergebenen *TEvent*-Records auf den Wert *evKeyDown* gesetzt und der Code der Taste im Feld *KeyCode* gespeichert. Wird kein Tastendruck ermittelt, enthält *What* den Wert *evNothing*. *TProgram.GetEvent* ruft diese Routine auf.

Querverweis *TProgram.GetEvent*, *evXXXX*-Konstanten, *TView.HandleEvent*

GetMouseEvent-Prozedur **DRIVERS**

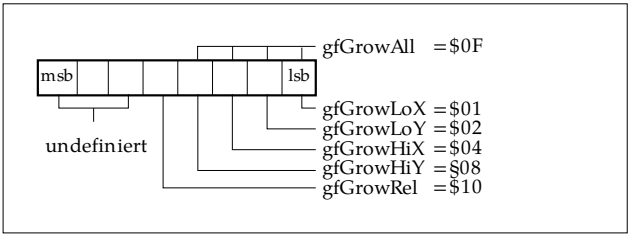
Deklaration
```
procedure GetMouseEvent(var Event: TEvent);
```

Funktion Überprüft, ob ein Mausereignis zur Bearbeitung ansteht. Sollte dies der Fall sein, wird *Event.What* auf den Wert *evMouseDown*, *evMouseUp*, *evMouseMove* oder *evMouseAuto*, und *Event.Double* auf True oder False gesetzt. In *Event.Buttons* wird entweder *mbLeftButton* oder *mbRightButton*, in *Event.Where* die globalen Koordinaten der Position des Mauszeigers gespeichert. Diese Position bezieht sich auf das Koordinatensystem von *TApplication*. Ist kein Mausereignis verfügbar, speichert *Event.What* den Wert *evNothing*. *TProgram.GetEvent* ruft diese Routine auf.

Querverweis *TProgram.GetEvent*, *evXXXX*-Konstanten, *Txxxx.HandleEvent*

gfXXXX-Konstanten **VIEWS**

Funktion Jedes *View*-Objekt definiert das Feld *GrowMode*, dessen Wert bestimmt, wie dieses Objekt reagiert, wenn die Größe der Gruppe, zu der es gehört, geändert wird.



Werte Die Konstanten, deren Namen mit »*gf*« beginnen, dienen dazu, dem Feld *GrowMode* die gewünschten Werte zuzuweisen, und sind folgendermaßen definiert:

Konstante	Wert	Bedeutung
gfGrowLoX	$01	Wenn dieses Bit gesetzt ist, bewahrt das *View*-Objekt eine konstante Distanz von der linken Seite der Gruppe.
gfGrowLoY	$02	Wenn dieses Bit gesetzt ist, bewahrt das *View*-Objekt eine konstante Distanz von der oberen Seite der Gruppe.

Konstante	Wert	Bedeutung
gfGrowHiX	$04	Wenn dieses Bit gesetzt ist, bewahrt das *View*-Objekt eine konstante Distanz von der rechten Seite der Gruppe.
gfGrowHiY	$08	Wenn dieses Bit gesetzt ist, bewahrt das *View*-Objekt eine konstante Distanz von der unteren Seite der Gruppe.
gfGrowAll	$0F	Wenn dieses Bit gesetzt ist, ändert das *View*-Objekt seine Größe nicht und bewegt sich zusammen mit der unteren rechten Ecke der Gruppe.
gfGrowRel	$10	Das *View*-Objekt ändert seine Größe im gleichen Verhältnis, in dem auch die Gruppe ihre Größe ändert. Diese Option sollten Sie nur bei *TWindow* und dessen Nachkommen verwenden, wenn solche Objekte zur Gruppe Arbeitsfläche gehören. Das Fenster bewahrt dann seine relative Größe, wenn zwischen 25- und 43/50-Zeilen-Modus umgeschaltet wird. Verwenden Sie diese Option nicht für *View*-Objekte eines Fensters.

Beachten Sie, daß sich *LoX* auf den linken, *LoY* auf den oberen, *HiX* auf den rechten und *HiY* auf den unteren Rand bezieht.

Querverweis *TView.GrowMode*

hcXXXX-Konstanten VIEWS

Funktion Über diese Konstanten kann der Programmkontext ermittelt werden, aus dem heraus der Benutzer Hilfe angefordert hat. Für eigene Zwecke reserviert Turbo Vision die Werte im Bereich 0 bis 999, der Bereich von 1000 bis 65535 steht dem Programmierer zur Verfügung.

Werte

Die folgenden Konstanten sind in Turbo Vision definiert:

Konstante	Wert	Bedeutung
hcNoContext	0	keine eigene Hilfenummer definiert
hcDragging	1	Position des Objekts wird verändert

Der voreingestellte Wert des Feldes *TView.HelpCtx* ist *hcNoContext*, was bedeutet, daß dieses Objekt keine eigene Hilfenummer definiert hat. *TView.GetHelpCtx* gibt immer dann *hcDragging* zurück, wenn das Objekt über den Bildschirm bewegt wird, was daran erkennbar ist, daß das Bit *sfDragging* im Feld *State* gesetzt ist.

Querverweis *TView.HelpCtx*, *TStatusLine.Update*

HideMouse-Prozedur DRIVERS

Deklaration

```
procedure HideMouse;
```

Funktion Nach der Ausführung von *InitEvents* ist der Mauszeiger sichtbar. Diese Routine macht den Mauszeiger unsichtbar und inkrementiert einen internen Zähler. *ShowMouse* stellt den Mauszeiger wieder dar und dekrementiert den Zähler.

Querverweis *InitEvents*, *DoneEvents*, *ShowMouse*

HiResScreen-Variable DRIVERS

Deklaration

```
HiResScreen: Boolean;
```

Funktion Wird von *InitVideo* auf True gesetzt, wenn der Bildschirmadapter (EGA oder VGA) den 43- oder 50-Zeilen-Modus unterstützt, andernfalls auf False.

Querverweis *InitVideo*

HistoryAdd-Prozedur HISTLIST

Deklaration

```
procedure HistoryAdd(Id: Byte; var Str: String);
```

Funktion Fügt den String *Str* der durch *Id* angesprochenen Eingabeaufzeichnungsliste hinzu.

HistoryBlock-Variable HISTLIST

Deklaration
```
HistoryBlock: Pointer = nil;
```

Funktion Die Prozedur *InitHistory* setzt diesen Zeiger auf den Puffer, der für die Speicherung von Eingabeaufzeichnungslisten benutzt wird. Die Größe dieses Puffers wird durch die Variable *HistorySize* festgelegt.

Querverweis *InitHistory, HistorySize*

HistoryCount-Funktion HISTLIST

Deklaration
```
function HistoryCount(Id: Byte): Word;
```

Funktion Gibt die Anzahl der Strings zurück, die in der über *Id* angesprochenen Eingabeaufzeichnungsliste gespeichert sind.

HistorySize-Variable HISTLIST

Deklaration
```
HistorySize: Word = 1024;
```

Funktion Gibt die Größe des Puffers an, der zur Speicherung von Eingabeaufzeichnungslisten verwendet wird. *InitHistory* reserviert einen Speicherbereich in dieser Größe. Dieser Wert kann vor dem Aufruf von *InitHistory*, darf aber danach nicht mehr geändert werden.

Querverweis *InitHistory, HistoryBlock*

HistoryStr-Funktion HISTLIST

Deklaration
```
function HistoryStr(Id: Byte; Index: Integer): String;
```

Funktion Gibt den String mit der Nummer *Index* in der Eingabeaufzeichnungsliste *Id* zurück.

HistoryUsed-Variable HISTLIST

Deklaration
```
HistoryUsed: Word = 0;
```

Funktion Diese Variable wird intern verwendet und speichert einen Offset in den Puffer, der die Eingabeaufzeichnungslisten enthält. Sie dürfen den Wert dieser Variablen nicht ändern.

InitEvents-Prozedur DRIVERS

Deklaration
```
procedure InitEvents;
```

Funktion Initialisiert die *Event*-Verwaltung von Turbo Vision und veranlaßt die Anzeige des Mauszeigers. Diese Prozedur wird von der Methode *TApplication.Init* aufgerufen.

Querverweis *TApplication.Init, DoneEvents*

InitHistory-Prozedur HISTLIST

Deklaration
```
procedure InitHistory;
```

Funktion Reserviert einen Bereich auf dem Heap, der zur Speicherung von Eingabeaufzeichnungslisten benutzt wird. Die Größe dieses Bereichs ist festgelegt durch den Wert der Variablen *HistorySize*; die Variable *HistoryBlock* zeigt auf den reservierten Bereich. Diese Prozedur wird von der Methode *TApplication.Init* aufgerufen.

Querverweis *TProgram.Init, DoneHistory*

InitMemory-Prozedur MEMORY

Deklaration
```
procedure InitMemory;
```

Funktion Initialisiert die Speicherverwaltung von Turbo Vision und die Überwachungsroutine, über die ermittelt werden kann, ob eine Speicheranforderung nur bei Verwendung des Sicherheitsbereichs erfüllt werden kann.

Querverweis *DoneMemory*

InitSysError-Prozedur DRIVERS

Deklaration
```
procedure InitSysError;
```

Funktion Übernimmt die Interruptvektoren $09, $1B, $21, $23 und $24, wodurch es Turbo Vision möglich ist, kritische Fehler abzufangen. Diese Prozedur wird von der Methode *TApplication.Init* aufgerufen.

Querverweis *DoneSysError*

InitVideo-Prozedur DRIVERS

Deklaration
```
procedure InitVideo;
```

Funktion Speichert den vorgefundenen Bildschirmmodus in *StartupMode* und schaltet auf den Modus um, der in *ScreenMode* angegeben ist. Die Variablen *ScreenWidth*, *ScreenHeigth*, *HiResScreen*, *CheckSnow*, *ScreenBuffer* und *CursorLines* werden diesem Modus entsprechend gesetzt. Über *SetVideoMode* ist es möglich, auf einen anderen Bildschirmmodus zu schalten. Diese Prozedur wird von *TApplication.Init* aufgerufen.

Querverweis *DoneVideo*, *SetVideoMode*, *smXXXX*

kbXXXX-Konstanten DRIVERS

Funktion Turbo Vision definiert zwei Arten von Konstanten, die jeweils das Präfix *kb* aufweisen.

Werte Die folgenden Konstanten dienen dazu, den Tastaturstatus, der in einem Word an der absoluten Adresse $40:$17 gespeichert ist, zu ermitteln, wie zum Beispiel

```
var
  ShiftState: Byte absolute $40:$17;
  ...
if ShiftState and kbAltShift <> 0 then AltKeyDown;
```

Konstante	Wert	Bedeutung
kbRightShift	$0001;	rechte Shift-Taste gedrückt
kbLeftShift	$0002;	linke Shift-Taste gedrückt
kbCtrlShift	$0004;	Crtl-Taste gedrückt
kbAltShift	$0008;	Alt-Taste gedrückt
kbScrollState	$0010;	Scroll-Lock aktiviert
kbNumState	$0020;	Num-Lock aktiviert
kbCapsState	$0040;	Caps-Lock aktiviert
kbInsState	$0080;	Insert aktiviert

Die folgenden Konstanten definieren Tastencodes und können dazu verwendet werden, das Feld *TEvent.KeyCode* auszuwerten, wenn ein Ereignis des Typs *evKeyDown* vorliegt:

`[Alt][Buchstabe]`-*Tastenkombinationen*

Konstante	Wert	Konstante	Wert
kbAltA	$1E00;	kbAltN	$3100
kbAltB	$3000;	kbAltO	$1800;
kbAltC	$2E00;	kbAltP	$1900
kbAltD	$2000;	kbAltQ	$1000;
kbAltE	$1200;	kbAltR	$1300;
kbAltF	$2100;	kbAltS	$1F00;
kbAltG	$2200;	kbAltT	$1400;
kbAltH	$2300;	kbAltU	$1600;
kbAltI	$1700;	kbAltV	$2F00;
kbAltJ	$2400;	kbAltW	$1100;
kbAltK	$2500;	kbAltX	$2D00;
kbAltL	$2600;	kbAltY	$1500;
kbAltM	$3200;	kbAltZ	$2C00;

Tastaturcodes von Spezialtasten

Konstante	Wert	Konstante	Wert
kbAltEqual	$8300;	kbEnd	$4F00
kbAltMinus	$8200;	kbEnter	$1C0D;
kbAltSpace	$0200;	kbEsc	$011B;
kbBack	$0E08;	kbGrayMinus	$4A2D;
kbCtrlBack	$0E7F;	kbGrayPlus	$4E2B;
kbCtrlDel	$0600;	kbHome	$4700
kbCtrlEnd	$7500;	kbIns	$5200;
kbCtrlEnter	$1C0A;	kbLeft	$4B00;
kbCtrlHome	$7700;	kbNoKey	$0000;
kbCtrlIns	$0400;	kbPgDn	$5100;
kbCtrlLeft	$7300;	kbPgUp	$4900;
kbCtrlPgDn	$7600;	kbRight	$4D00;
kbCtrlPgUp	$8400;	kbShiftDel	$0700;
kbCtrlPrtSc	$7200;	kbShiftIns	$0500;
kbCtrlRight	$7400;	kbShiftTab	$0F00;
kbDel	$5300;	kbTab	$0F09;
kbDown	$5000;	kbUp	$4800;

`[Alt][Ziffer]`-*Tastenkombinationen*

Konstante	Wert	Konstante	Wert
kbAlt1	$7800;	kbAlt6	$7D00;
kbAlt2	$7900;	kbAlt7	$7E00;
kbAlt3	$7A00;	kbAlt8	$7F00;
kbAlt4	$7B00;	kbAlt9	$8000;
kbAlt5	$7C00;	kbAlt0	$8100;

Tastaturcodes von Funktionstasten

Konstante	Wert	Konstante	Wert
kbF1	$3B00;	kbF6	$4000;
kbF2	$3C00;	kbF7	$4100;
kbF3	$3D00;	kbF8	$4200;
kbF4	$3E00;	kbF9	$4300;
kbF5	$3F00;	kbF10	$4400;

Tastaturcodes von Funktionstasten mit `[Shift]`

Konstante	Wert	Konstante	Wert
kbShiftF1	$5400;	kbShiftF6	$5900;
kbShiftF2	$5500;	kbShiftF7	$5A00;
kbShiftF3	$5600;	kbShiftF8	$5B00;
kbShiftF4	$5700;	kbShiftF9	$5C00;
kbShiftF5	$5800;	kbShiftF10	$5D00;

Tastaturcodes von Funktionstasten mit `[Ctrl]`

Konstante	Wert	Konstante	Wert
kbCtrlF1	$5E00;	kbCtrlF6	$6300;
kbCtrlF2	$5F00;	kbCtrlF7	$6400;
kbCtrlF3	$6000;	kbCtrlF8	$6500;
kbCtrlF4	$6100;	kbCtrlF9	$6600;
kbCtrlF5	$6200;	kbCtrlF10	$6700;

Tastaturcodes von Funktionstasten mit `[Alt]`

Konstante	Wert	Konstante	Wert
kbAltF1	$6800;	kbAltF6	$6D00;
kbAltF2	$6900;	kbAltF7	$6E00;
kbAltF3	$6A00;	kbAltF8	$6F00;
kbAltF4	$6B00;	kbAltF9	$7000;
kbAltF5	$6C00;	kbAltF10	$7100;

Querverweis *evKeyDown, GetKeyEvent*

LongDiv-Funktion

OBJECTS

Deklaration
```
function LongDiv(X: LongInt; Y:Integer): Integer;
inline($59/$58/$5A/$F7/$F9);
```

Funktion Diese Routine gibt das Ergebnis der Division eines LongInt-Wertes durch einen Integer-Wert zurück.

LongMul-Funktion

OBJECTS

Deklaration
```
function LongMul(X, Y:Integer): LongInt;
inline($5A/$58/$F7/$EA);
```

Funktion Diese Routine gibt in einem LongInt-Wert das Ergebnis der Multiplakation zweier Integer-Werte zurück.

LongRec-Typ

OBJECTS

Deklaration
```
LongRec = record
  Lo, Hi: Word;
end;
```

Funktion Dieser Recordtyp erlaubt, auf das niederwertige (*Lo*) und das höherwertige (*Hi*) Word eines LongInt-Wertes zuzugreifen.

LowMemory-Funktion

MEMORY

Deklaration
```
function LowMemory: Boolean;
```

Funktion Gibt True zurück, wenn eine Speicheranforderung, zum Beispiel durch eine Konstruktor-Methode, nur unter Verwendung des Sicherheitsbereichs erfüllt werden kann. Die Variable *LowMemorySize* legt die Größe dieses Sicherheitsbereichs fest.

Querverweis Kapitel 7 »*Sichere Programme schreiben*«, *InitMemory*, *TView.Valid*, *LowMemSize*

LowMemSize-Variable

MEMORY

Deklaration
```
LowMemSize: Word = 4096 div 16;
```

Funktion Gibt in Blöcken von jeweils 16 Bytes die Größe des Sicherheitsbereichs an. Die voreingestellte Größe entspricht einem Minimum, das für die meisten Programme ausreichend ist. Sie können diesen Wert jedoch größer ansetzen.

Querverweis *InitMemory*, Kapitel 7 »*Sichere Programme schreiben*«, *TView.-Valid*, *LowMemory*

MaxBufMem-Variable

MEMORY

Deklaration
```
MaxBufMem: Word = 65536 div 16;
```

Funktion Gibt in Blöcken von jeweils 16 Bytes die maximale Gesamtgrö-

ße des Speicherbereichs an, der den *Draw*-Puffern zugewiesen werden kann.

Querverweis *GetBufMem*, *FreeBufMem*

MaxCollectionSize-Variable OBJECTS

Deklaration
```
MaxCollectionSize = 65520 div SizeOf(Pointer);
```

Funktion Legt die maximale Zahl von Einträgen fest, die eine Kollektion aufnehmen kann. Diese Größe ergibt sich daraus, daß ein Zeiger 4 Bytes belegt, und die Gesamtzahl der Zeiger in ein 64 KByte großes Speichersegment passen muß.

MaxViewWidth-Konstante VIEWS

Deklaration
```
MaxViewWidth = 132;
```

Funktion Bestimmt die maximale Breite eines *View*-Objekts.

Querverweis *TView.Size*

mbXXXX-Konstanten DRIVERS

Funktion Diese Konstanten werden dazu benutzt, das Feld *TEvent.Buttons* eines Ereignisrecords des Typs *evMouse* abzufragen. Zum Beispiel durch:

```
if (Event.What = evMouseDown) and
  (Event.Buttons = mbLeftButton) then LeftButtonDown;
```

Werte In Turbo Vision sind die folgenden Konstanten definiert:

Konstante	Wert	Bedeutung
mbLeftButton	$01	Linke Maustaste gedrückt
mbRightButton	$02	Rechte Maustaste gedrückt

Querverweis *GetMouseEvent*

MemAlloc-Funktion MEMORY

Deklaration
```
function MemAlloc(Size: Word): Pointer;
```

Funktion Reserviert *Size* Bytes Speicher auf dem Heap und gibt einen Zeiger auf diesen Bereich zurück. Steht nicht mehr genügend Speicher zur Verfügung, gibt die Funktion **nil** zurück. Im Gegensatz zu den Standardprozeduren *New* und *GetMem* läßt

MemAlloc nicht zu, daß der Sicherheitsbereich mitverwendet wird. Der über diese Funktion reservierte Speicher kann mittels der Standardprozedur *FreeMem* freigegeben werden.

Querverweis *New*, *GetMem*, *Dispose*, *FreeMem*, *MemAllocSeg*

MemAllocSeg-Funktion MEMORY

Deklaration
```
function MemAllocSeg(Size: Word): Pointer;
```

Funktion Reserviert einen an einer Segment-Grenze beginnenden Speicherbereich. Diese Funktion entspricht im wesentlichen *MemAlloc*, allerdings ist der Offsetwert des zurückgegebenen Zeigers garantiert 0.

Querverweis *MemAlloc*

MenuBar-Variable APP

Deklaration
```
MenuBar: PMenuView = nil;
```

Funktion Speichert einen Zeiger auf die Menüleiste (ein Nachkomme von *TMenuView*) eines Programms. *MenuBar* wird von *TProgram.InitMenuBar* (aufgerufen von *TProgram.Init*) initialisiert. Der Wert **nil** zeigt an, daß das Programm keine Menüleiste besitzt.

Message-Funktion VIEWS

Deklaration
```
function Message(Receiver: PView; What, Command: Word;
                 InfoPtr: Pointer): Pointer;
```

Funktion Belegt einen Ereignisrecord mit den übergebenen Werten *What* (in der Regel *evBroadcast*), *Command* und *InfoPtr* und ruft dann, sofern möglich, *Receiver^.HandleEvent* zur Abarbeitung dieses Ereignisses auf. *Message* gibt **nil** zurück, wenn *Receiver* **nil** ist oder das Ereignis nicht bearbeitet wurde. Wenn der von der *HandleEvent*-Methode zurückgegebene Ereignisrecord im Feld *What* den Wert *evNothing* enthält, ist das Ereignis bearbeitet, und *Message* gibt den Wert des Feldes *InfoPtr* dieses Ereignisrecords zurück. Dadurch kann ermittelt werden, welches Objekt das Ereignis bearbeitet hat. Die Methode *TScrollBar.ScrollDraw* sendet zum Beispiel folgende »Botschaft« an die Gruppe, zu der das Rollbalken-Objekt gehört:

```
Message(Owner, evBroadCast, cmScrollBarChanged, @Self);
```

Diese Anweisung stellt sicher, daß die betroffenen *ViewObjekte* immer dann neu ausgegeben werden, wenn das Feld *Value* des Rollbalkens seinen Wert ändert.

Querverweis *TView.HandleEvent*, *TEvent*, *cmXXXX*-Konstanten, *evXXXX*-Konstanten

MinWinSize-Variable VIEWS

Deklaration
```
MinWinSize: TPoint = (X: 16, Y: 6);
```

Funktion Legt die minimale Größe eines *TWindow*- oder davon abgeleiteten Objekts fest. Bei einem Aufruf von *TWindow.SizeLimits* werden diese Werte über den Parameter *Min* zurückgegeben. Eine Änderung der Werte in *MinWinSize* betrifft alle Fensterobjekte, sofern die Methode *SizeLimits* nicht überschrieben wurde.

Querverweis *TWindow.SizeLimits*

MouseButtons-Variable DRIVERS

Deklaration
```
MouseButtons: Byte;
```

Funktion Enthält den aktuellen Status der Mausknöpfe. Wann immer der Benutzer einen Mausknopf drückt oder losläßt, wird der entsprechende Wert in dieser Variablen abgelegt. Dieser Wert kann mittels der *mbXXXX*-Konstanten abgefragt werden.

Querverweis *mbXXXX*-Konstanten, *GetMouseEvent*

MouseEvents-Variable DRIVERS

Deklaration
```
MouseEvents: Boolean = False;
```

Funktion Wird auf True gesetzt, wenn eine Maus installiert ist und von *InitEvents* ermittelt werden kann. Ist *MouseEvents* False, werden alle Routinen, die Mausereignisse bearbeiten, übergangen.

Querverweis *GetMouseEvent*

MouseIntFlag-Variable DRIVERS

Deklaration
```
MouseIntFlag: Byte;
```

Funktion Diese Variable ist für interne Nutzung durch *View*-Objekte

vorgesehen, und wird entsprechend den auftretenden Mausereignissen gesetzt.

MouseReverse-Variable **DRIVERS**

Deklaration
```
MouseReverse: Boolean = False;
```

Funktion Ist diese Variable auf True gesetzt, wird im Feld *Buttons* eines Ereignisrecords der Wert *mbLeftButton* eingetragen, wenn der rechte, und *mbRightButton*, wenn der linke Mausknopf gedrückt wurde.

Querverweis *mbXXXX*-Konstanten, *GetMouseEvent*

MouseWhere-Variable **DRIVERS**

Deklaration
```
MouseWhere: TPoint;
```

Funktion Speichert die aktuelle Position des Mauszeigers. Diese Werte beziehen sich auf das Koordinatensystem von *TApplication*, also auf die Breite und Höhe des Bildschirms. Über *MakeLocal* erhalten Sie die entsprechenden fenster-bezogenen Koordinaten. *MouseWhere* wird zusammen mit anderen Werten an die *HandleEvent*-Methoden übergeben.

Querverweis *GetMouseEvent*, *Txxxx.GetEvent*, *MakeLokal*

MoveBuf-Prozedur **OBJECTS**

Deklaration
```
procedure MoveBuf(var Dest; var Source; Attr: Byte;
                  Count: Word);
```

Funktion Kopiert Text in einen Puffer, den dann die Methoden *TView.WriteBuf* und *TView.WriteLine* benutzen. *Dest* muß vom Typ *TDrawBuffer* oder einem entsprechenden **Array of** Word sein; *Source* ist ein **Array of** Byte. Die durch *Count* angegebene Anzahl Bytes werden aus *Source* in das niederwertige Byte des entsprechenden Word in *Dest* kopiert. In das höherwertige Byte des Word wird *Attr* eingetragen, bleibt aber unverändert, wenn *Attr* den Wert 0 hat.

Querverweis *TDrawBuffer*, *MoveChar*, *MoveCStr*, *MoveStr*

MoveChar-Prozedur

OBJECTS

Deklaration
```
procedure MoveChar(var Dest; C: Char; Attr: Byte;
                   Count: Word);
```

Funktion Kopiert Zeichen in einen Puffer, den dann die Methoden *TView.WriteBuf* und *TView.WriteLine* benutzen. *Dest* muß vom Typ *TDrawBuffer* oder einem entsprechenden **Array of** Word sein. Ist der ordinale Wert des Zeichens *C* größer 0, wird das jeweilige niederwertige Byte in *Count* Word von *Dest* auf *C* gesetzt. In das höherwertige Byte des Word wird *Attr* eingetragen, es bleibt aber unverändert, wenn *Attr* den Wert 0 hat.

Querverweis *TDrawBuffer, MoveBuf, MoveCStr, MoveStr*

MoveCStr-Prozedur

OBJECTS

Deklaration
```
procedure MoveCStr(var Dest; Str: String; Attr: Word);
```

Funktion Kopiert den String *Str* in einen Puffer, den dann die Methoden *TView.WriteBuf* und *TView.WriteLine* benutzen. *Dest* muß vom Typ *TDrawBuffer* oder einem entsprechenden **Array of** Word sein. Die Zeichen in *Str* werden in das niederwertige Byte des entsprechenden Word in *Dest* kopiert. Das höherwertige Byte wird entweder auf *Lo*(*Attr*) oder *Hi*(*Attr*) gesetzt. Im String enthaltene Tilde-Zeichen (~) schalten zwischen den beiden in *Attr* angegebenen Attributen um.

Querverweis *TDrawBuffer, MoveBuf, MoveChar, MoveStr*

MoveStr-Prozedur

OBJECTS

Deklaration
```
procedure MoveStr(var Dest; Str: String; Attr: Byte);
```

Funktion Kopiert den String *Str* in einen Puffer, den dann die Methoden *TView.WriteBuf* und *TView.WriteLine* benutzen. *Dest* muß vom Typ *TDrawBuffer* oder einem entsprechenden **Array of** Word sein. Die Zeichen in *Str* werden in das niederwertige Byte des entsprechenden Word in *Dest* kopiert. In das höherwertige Byte des Word wird *Attr* eingetragen, bleibt aber unverändert, wenn *Attr* den Wert 0 hat.

Querverweis *TDrawBuffer, MoveBuf, MoveChar, MoveCStr*

NewItem-Funktion

DIALOGS

Deklaration
```
function NewItem(Name, Param: TMenuStr; KeyCode: Word;
                 Command: Word; AHelpCtx: Word;
                 Next: PMenuItem): PMenuItem;
```

Funktion Reserviert Speicher für einen *TMenuItem*-Record, der eine Menüoption enthält, und gibt einen Zeiger darauf zurück. Der Parameter *Name* darf kein leerer String und *Command* darf nicht 0 sein. Aufrufe der Funktionen *NewItem*, *NewLine* und *NewSubMenu* können ineinander verschachtelt werden, wodurch mit einer Anweisung ein ganzes Menüsystem aufgebaut wird. Ein Beispiel dazu finden Sie in Kapitel 3, »Mit Turbo Vision programmieren«.

Querverweis *TApplication.InitMenuBar*, *TMenuView*, *NewLine*, *NewMenu*, *NewSubMenu*

NewLine-Funktion

MENUS

Deklaration
```
function NewLine(Next: PMenuItem): PMenuItem;
```

Funktion Reserviert Speicher für einen *TMenuItem*-Record, der für eine Trennungslinie zwischen zwei Menüoptionen steht, und gibt einen Zeiger darauf zurück.

Querverweis *TApplication.InitMenuBar*, *TMenuView*, *NewMenu*, *NewSubMenu*, *NewItem*

NewMenu-Funktion

MENUS

Deklaration
```
function NewMenu(Items: PMenuItem): PMenu;
```

Funktion Reserviert Speicher für einen *TMenu*-Record und gibt einen Zeiger darauf zurück. Die Felder *Items* und *Defaults* dieses Records werden auf den im Parameter *Items* angegebenen Wert gesetzt.

Querverweis *TApplication.InitMenuBar*, *TMenuView*, *NewLine*, *NewSubMenu*, *NewItem*

NewSItem-Funktion

DIALOGS

Deklaration
```
function NewSItem(Str: String; ANext: PSItem): PSItem;
```

Funktion Reserviert Speicher für einen *TSItem*-Record und gibt einen Zeiger darauf zurück. Das Feld *Value* dieses Records wird auf

NewStr(Str), das Feld *Next* auf *ANext* gesetzt. Diese Funktion ermöglicht zusammen mit dem Recordtyp *TSItem*, eine einfach verkettete Liste von Strings aufzubauen. Ein Beispiel dazu finden Sie in Kapitel 5, »View-Objekte«.

NewStatusDef-Funktion

MENUS

Deklaration
```
function NewStatusDef(AMin, AMax: Word;
                      AItems: PStatusItems;
                      ANext: PStatusDef): PStatusDef;
```

Funktion Reserviert Speicher für einen *TStatusDef*-Record und gibt einen Zeiger darauf zurück. Die übergebenen Parameter werden in diesem Record abgelegt. Aufrufe der Funktionen *NewStatusDef* und *NewStatusKey* können ineinander verschachtelt werden, wodurch mit einer Anweisung die gesamte Stauszeile aufgebaut wird. Ein Beispiel dazu finden Sie in Kapitel 3, »Mit Turbo Vision programmieren«.

Querverweis *TApplication.InitStatusLine*, *TStatusLine*, *NewStatusKey*

NewStatusKey-Funktion

MENUS

Deklaration
```
function NewStatusKey(AText: String; AKeyCode: Word;
                      ACommand: Word;
                      ANext: PStatusItem): PStatusItem;
```

Funktion Reserviert Speicher für einen *TStatusItem*-Record und gibt einen Zeiger darauf zurück. Die übergebenen Parameter werden in diesem Record abgelegt. Ist *AText* ein leerer String, wird diese Option nicht ausgegeben. Dennoch ist das über *AKeyCode* definierte Tastenkürzel mit *ACommand* verbunden.

Querverweis *TApplication.InitStatusLine*, *TStatusLine*, *NewStatusDef*

NewStr-Funktion

OBJECTS

Deklaration
```
function NewStr(S: String): PString;
```

Funktion Ist der String *S* leer, gibt *NewStr* **nil** zurück, andernfalls reserviert diese Funktion *Length*(*S*) + 1 Bytes Speicher, kopiert *S* in diesen Bereich und gibt einen Zeiger darauf zurück. Der mittels dieser Funktion reservierte Speicher darf nur über *DisposeStr* wieder freigegeben werden.

Querverweis *DisposeStr*

NewSubMenu-Funktion

MENUS

Deklaration

```
function NewSubMenu(Name: TMenuStr; AHelpCtx: Word;
                    SubMenu: PMenu;
                    Next: PMenuItem): PMenuItem;
```

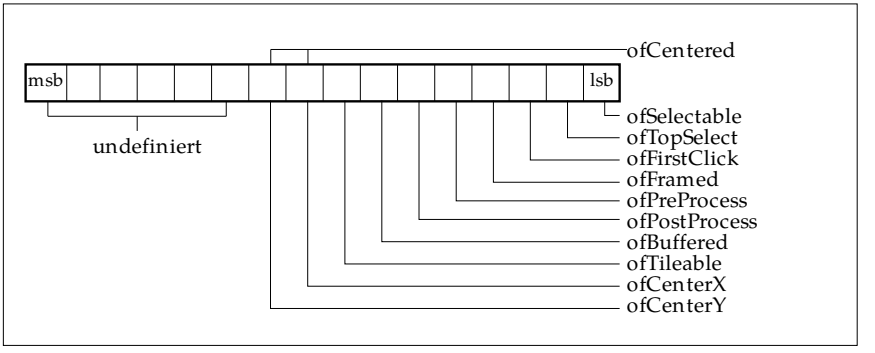
Funktion Reserviert Speicher für einen *TMenuItem*-Record, der eine Option in einem Submenü enthält, und gibt einen Zeiger darauf zurück.

Querverweis *TApplication.InitMenuBar*, *TMenuView*, *NewItem*, *NewLine*

ofXXXX-Konstanten

VIEWS

Funktion Jedes *View*-Objekt definiert das Feld *Options*, dessen einzelne Bits die Eigenschaften dieses Objekts bestimmen. Ist ein Bit gesetzt, verfügt das Objekt über die betreffende Eigenschaft.



Werte Die Konstanten, deren Namen mit »*of*« beginnen, dienen dazu, dem Feld *Options* die gewünschten Werte zuzuweisen, und sind folgendermaßen definiert:

Konstante	Wert	Bedeutung
ofSelectable	$0001	Ist dieses Bit gesetzt, kann der Benutzer das Objekt mit der Maus selektieren. Gehört das Objekt zu einer Gruppe, ist dies zusätzlich auch mittels [Tab] möglich. Wenn Sie etwas nur zur Information auf dem Bildschirm ausgeben, soll der Benutzer ein solches Objekt nicht selektieren können. Statischer Text oder

Konstante	Wert	Bedeutung
		Rahmen sind in der Regel nicht selektierbar.
ofTopSelect	$0002	Wenn selektiert, wird das Objekt in den Vordergrund versetzt. Gedacht ist diese Option vor allem für Fenster, und sollte für Objekte, die einer Gruppe angehören, nicht genutzt werden.
ofFirstClick	$0004	Der Mausklick, der dieses *View*-Objekt selektiert, wird an das Objekt selbst weitergereicht. Wird zum Beispiel ein Schalter gedrückt und dieses Bit ist gesetzt, löst ein Mausklick sowohl die Selektion des Schalters als auch die damit verbundene Aktion aus. Andere Objekte dagegen sollen nur selektiert werden können und keine weitere Aktion auslösen.
ofFramed	$0008	Diese Bit ist bei *TWindow*-Objekten gesetzt, wenn dieses Fenster mit einem Rahmen versehen werden soll, in allen anderen Fällen (und Objekten) ist es gelöscht.
ofPreProcess	$0010	Gesetzt, wenn ein fokussiertes *Event* zuerst an dieses Objekt geleitet werden muß, bevor es an das fokussierte *View*-Objekt weitergereicht wird.
ofPostProcess	$0020	Gesetzt, wenn fokussierte *Events*, die von dem fokussierten *View*-Objekt nicht abgearbeitet wurden, diesem Objekt zugeleitet werden sollen.
ofBuffered	$0040	Wird nur bei *TGroup*-Objekten benutzt. Über diese Option kann die Bildschirmausgabe einer Gruppe beschleunigt werden.Wenn eine Gruppe zum ersten Mal dargestellt wird, speichert sie automatisch auch den

Konstante	Wert	Bedeutung
		von ihr belegten Bildschirmbereich, sofern dieses Bit gesetzt und genügend Speicher verfügbar ist. Muß diese Gruppe ein weiteres Mal auf dem Bildschirm ausgegeben werden, kopiert die Gruppe den Puffer in den Bildschirmspeicher, anstatt die *Subviews* aufzufordern, sich selbst auszugeben. Wenn ein Aufruf von *New* oder *GetMem* nicht mehr genügend Speicher vorfindet, beginnt die Speicherverwaltung automatisch, die belegten Puffer wieder freizugeben, bis die Speicheranforderung erfüllt werden kann.
ofTileable	$0080	Gesetzt, wenn die Arbeitsfläche dieses Objekt zusammen mit anderen nebeneinander oder fächerartig hintereinander darstellen kann. Wird normalerweise nur bei *TWindow*-Objekten gesetzt.
ofCenterX	$0100	Wenn in die Gruppe eingefügt, wird das *View*-Objekt in X-Richtung zentriert.
ofCenterY	$0200	Wenn in die Gruppe eingefügt, wird das *View*-Objekt in Y-Richtung zentriert.
ofCentered	$0300	Wenn in die Gruppe eingefügt, wird das *View*-Objekt sowohl in X- als auch in Y-Richtung zentriert.

Querverweis *TView.Options*

PChar-Typ

OBJECTS

Deklaration
```
PChar = ^Char;
```

Funktion Definiert einen Zeiger auf ein Zeichen.

PositionalEvents-Variable VIEWS

Deklaration
```
PositionalEvents: Word = evMouse;
```

Funktion Definiert die Ereignisklassen, die als *Positionsereignisse* anzusehen sind. Über die Variablen *FocusedEvents* und *PositionalEvents* entscheidet *TGroup.HandleEvent*, in welcher Weise ein Ereignis an die zu dieser Gruppe gehörenden Objekte weitergegeben werden. Ist ein Ereignis weder den in *FocusedEvents* noch den in *PositionalEvents* definierten Ereignisklassen zuzuordnen, wird es als ein Ereignis des Typs *evBroadCast* behandelt.

Querverweis *FocusedEvents*, *TGroup.HandleEvent*, *TEvent*, *evXXXX*-Konstanten

PrintStr-Prozedur DRIVERS

Deklaration
```
procedure PrintStr(S: String);
```

Funktion Gibt über die DOS-Funktion $40 den String *S* auf dem Standard-Ausgabegerät aus. Die Prozedur hat die gleiche Wirkung wie die Anweisung *Write(S)*, allerdings vermeidet *PrintStr*, daß die Runtime-Routinen zur Ein-/Ausgabe in das Programm eingebunden werden müssen.

PString-Typ OBJECTS

Deklaration
```
PString = ^String;
```

Funktion Definiert einen Zeiger auf einen String.

PtrRec-Typ OBJECTS

Deklaration
```
PtrRec = record
  Ofs, Seg: Word;
end;
```

Funktion Dieser Recordtyp ermöglicht den Zugriff auf den Offset- (*Ofs*) und den Segmentwert (*Seg*) eines Zeigers.

RegisterApp-Prozedur APP

Deklaration `procedure RegisterApp;`

Funktion Ruft für die in der Unit App definierten Objekte *TBackGround* und *TDeskTop* die Prozedur *RegisterTypes* auf. Damit können diese Objekte in einem Stream geschrieben und aus einem Stream gelesen werden.

Querverweis *TStreamRec, RegisterTypes*

RegisterDialogs-Prozedur DIALOGS

Deklaration `procedure RegisterDialogs;`

Funktion Ruft für jedes in der Unit *Dialogs* definierte Objekt die Prozedur *RegisterTypes* auf. Im Einzelnen betrifft das die Objekte *TDialog, TInputLine, TButton, TCluster, TRadioButtons, TCheckBoxes, TListBox, TStaticText, TParamText, TLabel* und *THistory*. Damit können diese Objekte in einen Stream geschrieben und aus einem Stream gelesen werden.

Querverweis *TStreamRec, RegisterTypes*

RegisterMenus-Prozedur MENUS

Deklaration `procedure RegisterMenus;`

Funktion Ruft für die in der Unit *Menus* definierten Objekte *TMenuBar, TMenuBox* und *TStatusLine* die Prozedur *RegisterTypes* auf. Damit können diese Objekte in einen Stream geschrieben und aus einem Stream gelesen werden.

Querverweis *TStreamRec, RegisterTypes*

RegisterObjects-Prozedur OBJECTS

Deklaration `procedure RegisterObjects;`

Funktion Ruft für die in der Unit *Objects* definierten Objekte *TCollection* und *TStringCollection* die Prozedur *RegisterTypes* auf. Damit können diese Objekte in einen Stream geschrieben und aus einem Stream gelesen werden.

Querverweis *TStreamRec, RegisterTypes*

RegisterTypes-Prozedur OBJECTS

Deklaration
```
procedure RegisterTypes(var S: TStreamRec);
```

Funktion Jedes Turbo-Vision-Objekt muß mittels dieser Prozedur registriert werden, bevor dessen Ein- und Ausgabe über einen Stream möglich ist. Die Prozedur *RegisterTypes* fügt bei einem Aufruf einen weiteren Eintrag in die einfach verkettete Liste von Records des Typs *TStreamRec* ein.

Querverweis *TStream.Get*, *TStream.Put*, *TStreamRec*

RegisterViews-Prozedur VIEWS

Deklaration
```
procedure RegisterViews;
```

Funktion Ruft für die in der Unit *Views* definierten Objekte *TView*, *TFrame*, *TScrollBar*, *TScroller*, *TListViewer*, *TGroup* und *TWindow* die Prozedur *RegisterTypes* auf. Damit können diese Objekte in einem Stream geschrieben und aus einen Stream gelesen werden.

Querverweis *TStreamRec*, *RegisterTypes*

RepeatDelay-Variable DRIVERS

Deklaration
```
RepeatDelay: Word = 8;
```

Funktion Definiert den Zeitraum (anzugeben in 1/18,2 Bruchteilen einer Sekunde) nach dessen Ablauf Turbo Vision beginnt, Ereignisse des Typs *evMouseAuto* zu generieren. Daraufhin wird im Abstand von 1/18,2 Sekunden jeweils ein solches Ereignis erzeugt.

Querverweis *DoubleDelay*

SaveCtrlBreak-Variable DRIVERS

Deklaration
```
SaveCtrlBreak: Boolean = False;
```

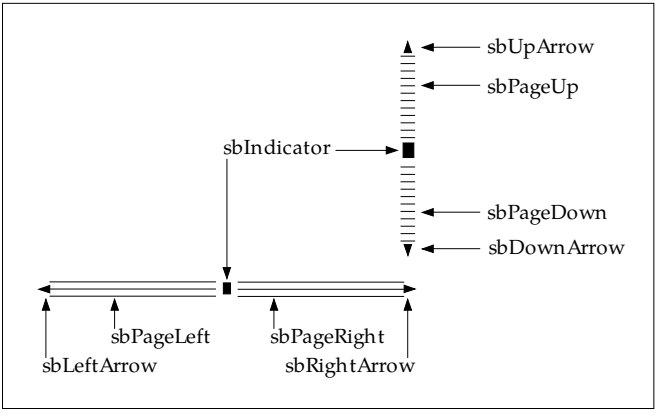
Funktion *InitSysError* sichert in dieser Variablen den vorgefundenen Status der *Ctrl-Break*-Überprüfung durch DOS, und schaltet diese Überprüfung ab. *DoneSysError* stellt den vorgefundenen Status wieder her.

Querverweis *InitSysError*, *DoneSysError*, *CtrlBreakHit*

sbXXXX-Konstanten

Funktion

Diese Konstanten bezeichnen die verschiedenen Bereiche eines Rollbalkens, die mit der Maus angeklickt werden können. Die Methode *TScrollBar.ScrollStep* konvertiert diese Konstanten in die tatsächlich verwendeten Werte für die Schrittweite, in der zu rollen ist. Obgleich definiert, wird die Konstante *sbIndicator* nie an *TScrollBar.ScrollStep* übergeben.



Werte

Konstante	Wert	Bedeutung
sbLeftArrow	0	Pfeilsymbol an der linken Seite des horizontalen Rollbalkens
sbRightArrow	1	Pfeilsymbol an der rechten Seite des horizontalen Rollbalkens
sbPageLeft	2	Bereich im horizontalen Rollbalken, über den nach links gerollt wird
sbPageRight	3	Bereich im horizontalen Rollbalken, über den nach rechts gerollt wird
sbUpArrow	4	Pfeilsymbol an der oberen Seite des vertikalen Rollbalkens
sbDownArrow	5	Pfeilsymbol an der unteren Seite des vertikalen Rollbalkens
sbPageUp	6	Bereich im vertikalen Rollbalken, über den nach oben geblättert wird

Konstante	Wert	Bedeutung
sbPageDown	7	Bereich im vertikalen Rollbalken, über den nach unten geblättert wird
sbIndicator	8	Positionsindikator auf den Rollbalken

Die folgenden Werte können der Methode *TWindow.StandardScrollBar* übergeben werden:

Konstante	Wert	Bedeutung
sbHorizontal	$0000	Rollbalken ist horizontal orientiert
sbVertical	$0001	Rollbalken ist vertikal orientiert
sbHandle-Keyboard	$0002	Rollbalken reagiert auf Tastaturereignisse

Querverweis *TScrollBar*, *TScrollBar.ScrollStep*

ScreenBuffer-Variable **DRIVERS**

Deklaration
```
ScreenBuffer: Pointer;
```

Funktion Dieser Zeiger auf den Bildschirmspeicher wird von *InitVideo* gesetzt.

Querverweis *InitVideo*

SreenHeight-Variable **DRIVERS**

Deklaration
```
ScreenHeight: Byte;
```

Funktion Definiert die Maximalzahl der auf dem Bildschirm darstellbaren Zeilen. Dieser Wert wird von *InitVideo* ermittelt.

Querverweis *InitVideo*, *SetVideoMode*, *ScreenWidth*

ScreenMode-Variable **DRIVERS**

Deklaration
```
ScreenMode: Word;
```

Funktion Diese Variable speichert den gerade aktiven Bildschirmmodus, der beim Start des Programms von *InitVideo* ermittelt wird. Er kann mittels *SetVideoMode* geändert werden kann. Die Modi,

die Turbo Vision unterstützt, sind in den *smXXXX*-Konstanten festgehalten.

Querverweis *InitVideo*, *SetVideoMode*, *smXXXX*-Konstanten

ScreenWidth-Variable **DRIVERS**

Deklaration
```
ScreenWidth: Byte;
```

Funktion Maximalzahl der auf dem Bildschirm darstellbaren Spalten. Dieser Wert wird von *InitVideo* ermittelt.

Querverweis *InitVideo*, *ScreenHeight*

SelectMode-Typ **VIEWS**

Deklaration
```
SelectMode = (NormalSelect, EnterSelect, LeaveSelect);
```

Funktion Wird intern von Turbo Vision benutzt.

Querverweis *TGroup.ExecView*, *TGroup.SetCurrent*

SetMemTop-Prozedur **Memory**

Deklaration
```
procedure SetMemTop(MemTop: Pointer);
```

Funktion Setzt die obere Grenze des Heap-Speichers auf den durch den Zeiger *MemTop* angegebenen Wert. Damit läßt sich der Speicherbedarf des Programmes reduzieren (und wieder erweitern), um Platz für andere Applikationen zu schaffen. Die folgenden Anweisungen

```
SetMemTop(HeapPtr);
PrintStr('Mit EXIT zum Programm zur¡ck');
SwapVectors;
Exec(GetEnv('COMSPEC'), '');
SwapVectors;
SetMemTop(HeapEnd);
```

zeigen, wie einfach Sie mittels dieser Prozedur aus einem Programm heraus die DOS-Shell aufrufen können, und dies auch dann, wenn das Programm den gesamten freien Speicher für sich als Heap reklamiert. Zunächst wird über *SetMemTop(HeapPtr)* die obere Grenze des Heap soweit als möglich heruntergesetzt, dann mit Hile von *GetEnv* ermittelt, in welchem Verzeichnis COMMAND.COM gespeichert ist. Darauf wird dieses Programm aufgerufen, und – wenn der Benutzer

EXIT eingegeben hat – die Heap-Grenze über *SetMemTopHeapEnd* wieder auf den alten Wert gesetzt.

SetVideoMode-Prozedur

DRIVERS

Deklaration
```
procedure SetVideoMode(Mode: Word);
```

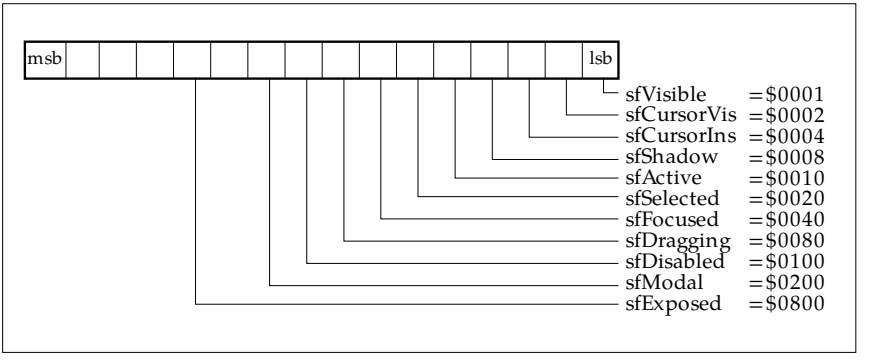
Funktion Setzt den Videomodus entsprechend dem in *Mode* übergebenen Wert (der einem der in den *smXXXX*-Konstanten gespeicherten Werte entsprechen muß). Sie sollten normalerweise diese Prozedur nicht direkt, sondern über *TApplication.SetScreenMode* aufrufen, da diese Methode auch die passende Farbpalette auswählt.

Querverweis *InitVideo*, *ScreenMode*, *smXXXX*-Konstanten, *TApplication.SetScreenMode*

sfXXXX-Konstanten

VIEWS

Funktion Jedes *View*-Objekt definiert das Feld *State*, dessen einzelne Bits über den aktuellen Status dieses Objekts Auskunft geben



Werte Die Konstanten, deren Namen mit »*sf*« beginnen, dienen dazu, dem Feld *State* die gewünschten Werte zuzuweisen. Sie sind folgendermaßen definiert:

Konstante	Wert	Bedeutung
sfVisible	$0001	Gesetzt, wenn das *View*-Objekt auf dem Bildschirm dargestellt ist.
sfCursorVis	$0002	Gesetzt, wenn der Cursor sichtbar ist.

Konstante	Wert	Bedeutung
sfCursorIns	$0004	Gesetzt wenn der Cursor als Block dargestellt wird. Per Voreinstellung ist dieses Bit gelöscht, was bedeutet, daß der Cursor – wenn sichtbar – als Unterstrich erscheint.
sfShadow	$0008	Gesetzt, wenn das *View*-Objekt mit einem Schatten versehen ist.
sfActive	$0010	Gesetzt, wenn das *View*-Objekt gerade aktiviert ist.
sfSelected	$0020	Gesetzt, wenn das Objekt die *selektierte* Subview der Gruppe ist, der es angehört. In einer Gruppe kann immer nur ein Objekt selektiert sein, und jedes *TGroup*-Objekt speichert in seinem Feld *Current* einen Zeiger auf diese aktuell selektierte Subview. Diese kann natürlich wiederum eine Gruppe mit einer selektierten Subview sein. Die so entstehende Kette von Verweisen über die *Current*-Zeiger beschreibt den Selektierungspfad, der in der modalen Gruppe beginnt und mit der letzten selektierten Subview, der *fokussierten* Subview, endet. Über diesen Selektierungspfad gelangen zum Beispiel Ereignisse zu dem adressierten Objekt.
sfFocused	$0040	Gesetzt, wenn das Objekt fokussiert ist. Dieses Objekt steht dann am Ende des Selektierungspfads (siehe auch: *sfSelected*).
sfDragging	$0080	Gesetzt, wenn das Objekt gerade über den Bildschirm bewegt wird.
sfDisabled	$0100	Gesetzt, wenn das *View*-Objekt gerade deaktiviert ist. Ein solches Objekt ignoriert alle Ereignisse, die ihm übergeben werden.

Konstante	Wert	Bedeutung
sfModal	$0200	Gesetzt, wenn das Objekt modal ist. In einem Turbo-Vision-Programm ist immer genau ein *View*-Objekt modal, in der Regel ein *TMain*- oder *TDialog*-Objekt, die beide Nachkommen von*TGroup* sind. Ein modales Objekt beschränkt die Reichweite von Ereignissen auf die eigenen Subviews, darin eingeschlossen auch deren Subviews, und zwar solange, bis der Befehl *cfTerminateModal* erkannt wird.
sfDefault	$0400	Wird nur bei *TButton*-Objekten beachtet und ist gesetzt, wenn der Aktionsschalter voreingestellt ist, also mittels [Enter] aktiviert werden kann.
sfExposed	$0800	Gesetzt, wenn das Objekt eine Gruppe ist, die direkt oder indirekt zu *TApplication* gehört. Wird zusammen mit *TView.Exposed* ausgewertet, und erlaubt, *TView.DrawView* zu bestimmen, welche *View*-Objekte neu ausgegeben werden müssen.

Querverweis *TView.State*

ShadowAttr-Variable VIEWS

Deklaration
```
ShadowAttr: Byte = $08;
```

Funktion Bestimmt die Farbe zur Darstellung des Schatteneffekts bei solchen *View*-Objekten, in deren Feld *State* das Bit *sfShadow* gesetzt ist. Dieser Schatten ist ein schmaler, dunkler Bereich am rechten und unteren Rand eines Objekts.

Querverweis *ShadowSize*

ShadowSize-Variable VIEWS

Deklaration
```
ShadowSize: TPoint = (X: 2, Y:1);
```

Funktion Bestimmt die Größe des Schattens bei solchen *View*-Objekten,

in deren Feld *State* das Bit *sfShadow* gesetzt ist. Dieser Schatten ist ein schmaler, dunkler Bereich am rechten Rand, wo er per Voreinstellung zwei Spalten breit ist, und unteren Rand, wo er ebenfalls per Voreinstellung eine Zeile einnimmt. *TProgram.InitScreen* initialisiert *ShadowSize* in Abhängigkeit vom Bildschirmmodus. Ist dieser *smMono*, wird *ShadowSize* auf (0, 0), andernfalls auf (2, 1) gesetzt, es sei denn, *smFont8x8* (43- oder 50-Zeilen-Modus) ist gewählt. Dann wird *ShadowSize* mit den Werten (1, 1) belegt.

Querverweis *TProgram.InitScreen, ShadowAttr*

ShowMarkers-Variable DRIVERS

Deklaration
```
ShowMarkers: Boolean;
```

Funktion Wird benutzt, um anzugeben, ob fokussierte Schalter mit besonderen Zeichen versehen werden sollen. *TProgram.InitScreen* setzt *ShowMarkers* auf True, wenn ein monochromer Bildschirmmodus ermittelt wird, andernfalls auf False. Sie können diese Variable jedoch auf True setzen, wenn diese Zeichen auch auf einem Schwarzweiß- oder Farbbildschirm ausgegeben werden sollen.

Querverweis *TProgram.InitScreen, SpecialChars*

ShowMouse-Prozedur DRIVERS

Deklaration
```
procedure ShowMouse;
```

Funktion Dekrementiert einen internen Zähler (den die Prozedur *HideMouse* inkrementiert) und gibt, sollte dieser Zähler dann auf 0 stehen, den Mauszeiger auf dem Bildschirm aus.

Querverweis *HideMouse, InitEvents*

smXXXX-Konstanten DRIVERS

Funktion Diese Konstanten werden dazu benutzt, über *SetVideoMode* einen anderen Bildschirmmodus einzustellen.

Werte Die folgenden Bildschirmmodi sind in Turbo Vision definiert:

Konstante	Wert	Bedeutung
smBW80	$0002	Schwarzweißmodus auf einem Farbadapter
smCO80	$0003	Farbmodus
smMono	$0007	Monochromer Modus
smFont8x8	$0100	43-Zeilen- oder 50-Zeilen-Modus

Querverweis *SetVideoMode*, *ScreenMode*

SpecialChars-Variable

VIEWS

Deklaration
```
SpecialChars: array[0..5] of Char =
                    (#175, #174, #26,#27, ' ', ' ');
```

Funktion Definiert die Zeichen, die bei einem monochromen Bildschirmmodus dazu benutzt werden, den fokussierten Schalter zu markieren. Diese Zeichen werden nur dann ausgegeben, wenn die Variable *ShowMarkers* True ist.

Querverweis *ShowMarkers*

stXXXX-Konstanten

OBJECTS

Funktion Turbo Vision definiert zwei Gruppen von Konstanten, deren Name mit *st* beginnt, und die beide von *TStream*-Objekten verwendet werden.

Werte Die folgenden Konstanten nutzen *TDosStream* und *TBufStream*, um den Zugriffsmodus auf eine Datei festzulegen:

Konstante	Wert	Bedeutung
stCreate	$3C00	Neue Datei anlegen
stOpenRead	$3D00	Existierende Datei nur zum Lesen öffnen
stOpenWrite	$3D01	Existierende Datei nur zum Schreiben öffnen
stOpen	$3D02	Existierende Datei zum Lesen und zum Schreiben öffnen

Die folgenden Konstanten dienen dazu, Fehler in einem Stream zu charakterisieren. *TStream.Error* legt in einem solchen Fall den entsprechenden Wert im Feld *TStream.ErrorInfo* ab.

Konstante	Wert	Bedeutung
stOK	0	Kein Fehler aufgetreten
stError	-1	Fehler beim Dateizugriff
stInitError	-2	Fehler bei der Stream-Initialisierung
stReadError	-3	Versuch, über das Stream-Ende hinaus zu lesen
stWriteError	-4	Fehler beim Schreiben in den Stream
stGetError	-5	*Get* mit nicht registriertem Objekttyp aufgerufen
stPutError	-6	*Put* mit nicht registriertem Objekttyp aufgerufen

Querverweis *TStream*

StartupMode **DRIVERS**

Deklaration
```
StartupMode: Word;
```

Funktion In dieser Variablen speichert *InitVideo* den Bildschirmmodus ab, den das Programm vorfindet. Vor Beendigung des Programms überprüft *DoneVideo*, ob inzwischen der Modus verändert wurde, und schaltet gegebenenfalls wieder auf den alten Modus um.

Querverweis *ScreenMode*, *SetVideoMode*, *InitVideo*, *DoneVideo*.

StatusLine-Variable **APP**

Deklaration
```
StatusLine: PStatusLine = nil;
```

Funktion Diese Variable wird von *TProgram* und *TApplication* benutzt, und speichert einen Zeiger auf die Statuszeile des Programms. Der Wert **nil** zeigt an, daß das Programm keine Statuszeile besitzt.

Querverweis *InitStatusLine*

StreamError-Variable **OBJECTS**

Deklaration
```
StreamError: Pointer = nil;
```

Funktion Wenn ungleich **nil**, zeigt *StreamError* auf eine Prozedur, die von der *Error*-Methode eines Stream aufgerufen wurde, wenn in diesem Stream ein Fehler auftrat. Die Prozedur muß als **far**

deklariert und ihr einziger Parameter muß vom Typ *TStream* sein:

```
procedure MyStreamErrorProc(var S: TStream); far;
```

StreamError ermöglicht Ihnen, die Fehlerbehandlung aller Stream-Objekte zu übernehmen. Um die Reaktion auf Fehler eines bestimmten Stream-Objekttyps zu ändern, müssen Sie dagegen dessen Methode *Error* überschreiben.

SysColorAttr-Variable DRIVERS

Deklaration
```
SysColorAttr: Word = $4E4F;
```

Funktion Bestimmt die Farbe, in der Meldungen über kritische Fehler von der Routine *SystemError* ausgegeben werden. Wurde ein Monochrom-Bildschirmadapter erkannt, werden statt dieser Farben die in *SysMonAttr* definierten verwendet. Solche Fehlermeldungen werden in der Statuszeile ausgegeben, deren Inhalt wird zunächst gesichert, und – wenn es die Bedingungen erlauben – dann erneut dargestellt.

Querverweis *SystemError*, *SysMonAttr*

SysErrActive-Variable DRIVERS

Deklaration
```
SysErrActive: Boolean = False;
```

Funktion Zeigt an, ob die Fehlerbehandlungsroutine von Turbo Vision aktiv ist, und wird von *InitSysError* auf True gesetzt.

SysErrorFunc-Variable DRIVERS

Deklaration
```
SysErrorFunc: TSysErrorFunc = SystemError;
```

Funktion *SysErrorFunc* zeigt auf die in Turbo Vision definierte Fehlerbehandlungsroutine *SystemError*, die immer dann aufgerufen wird, wenn ein kritischer Fehler auftrat oder die logischen Laufwerke A und B in einem System mit nur einem Diskettenlaufwerk diesem neu zugeordnet werden. Der Funktiostyp *TSysErrorFunc* ist folgendermaßen definiert:

```
TSysErrorFunc = function(ErrorCode: Integer;
                         Drive: Byte): Integer;
```

ErrorCode ist ein Wert zwischen 0 und 15 (die folgende Tabelle zeigt diese Werte) und *Drive* ist die Nummer des Laufwerks (0=A, 1=B usw.).

Sie können eine eigene Routine über die Zuweisung an *SysErrorFunc* installieren, dürfen diese Routine aber auf keinen Fall in einem Overlay ablegen.
Die möglichen Werte für *ErrorCode* haben folgende Bedeutung:

Fehlercode	Bedeutung
0..12	Kritischer Fehler
13	Fehlerhafte Speicherkopie der FAT
14	Fehler beim Zugriff auf ein externes Gerät
15	Neue Zuordnung zwischen logischem und physikalischem Laufwerk erforderlich

Eine Fehlerbehandlungsroutine muß die folgenden Werte zurückgeben:

Wert	Bedeutung
0	Benutzer fordert einen erneuten Versuch, den Befehl auszuführen
1	Benutzer hat Abbruch des Befehls gewählt

Querverweis *SystemError, TSysErrorFunc, InitSysError*

SysMonoAttr-Variable

DRIVERS

Deklaration
```
SysMonoAttr: Word = $7070;
```

Funktion Bestimmt die Farbe, in der Meldungen über kritische Fehler von der Routine *SystemError* ausgegeben werden. Wurde ein Farb-Bildschirmadapter erkannt, werden statt dieser Farben die in *SysColorAttr* definierten verwendet. Solche Fehlermeldungen werden in der Statuszeile ausgegeben, deren Inhalt wird zunächst gesichert, und – wenn es die Bedingungen erlauben – dann erneut dargestellt.

Querverweis *SystemError, SysColorAttr*

SystemError-Funktion

DRIVERS

Deklaration
```
function SystemError(ErrorCode: Integer;
                     Drive: Byte): Integer;
```

Funktion Ist die in Turbo Vision definierte Fehlerbehandlungsroutine. Sie gibt die über *ErrorCode* ermittelte Meldung in der Statuszeile aus und verwendet dazu die in *SysColorAttr* oder *SysMonAttr* definierten Farben. Die folgende Tabelle zeigt die möglichen Fehlercodes und die entsprechenden Meldungen:

Fehlercode	Meldung
0	Disk is write-protected in drive X (Diskette im Laufwerk X ist schreibgeschützt)
1	Critical disk error on drive X (Kritischer Fehler im Laufwerk X aufgetreten)
2	Disk is not ready in drive X (Laufwerk X ist nicht bereit)
3	Critical disk error on drive X (Kritischer Fehler im Laufwerk X aufgetreten)
4	Data integrity error on drive X (Datenfehler in Laufwerk X)
5	Critical disk error on drive X (Kritischer Fehler im Laufwerk X aufgetreten)
6	Seek error on drive X (Zugriffsfehler in Laufwerk X)
7	Unknown media type in drive X (Unbekannter Medientyp in Laufwerk X)
8	Sector not found on drive X (Laufwerk X: Sektor nicht gefunden)
9	Printer out of paper (Kein Papier im Drucker)
10	Write fault on drive X (Schreibfehler auf Laufwerk X)
11	Read fault on drive X (Lesefehler auf Laufwerk X)
12	Hardware failure on drive X (Hardware-Fehler auf Laufwerk X)
13	Bad memory image of FAT detected (Fehlerhafte FAT ermittelt)
14	Device access error (Zugriffsfehler)
15	Insert diskette in drive X (Legen Sie Diskette in Laufwerk X ein)

Querverweis *SysColorAttr, SysMonoAttr, SysErrorFunc*

TByteArray-Typ

OBJECTS

Deklaration

```
TByteArray = array[0..32767] of Byte;
```

Funktion Dieser Datentyp wird von einigen Objekten (zum Beispiel *TStrListMaker*) dazu benutzt, über *GetMem* und einen Zeiger auf solch ein Array Speicher der jeweils notwendigen Größe zu reservieren.

Querverweis *TStrListMaker*

TCommandSet-Typ

VIEWS

Deklaration

```
TCommandSet = set of Byte;
```

Funktion

Dieser Datentyp ist dafür vorgesehen, einen Set von bis zu 256 Befehlen zusammenzustellen. Damit kann zum Beispiel eine *HandleEvent*-Methode leicht überprüfen, ob ein bestimmter Befehl einem gegebenen Kriterium entspricht oder nicht. Auch die in *TView* definierten Methoden *EnableCommands*, *DisableCommands*, *GetCommands* und *SetCommands* benutzen Argumente diesen Typs. Eine Variable des Typs *TCommandSet* kann wie ein Pascal-*Set* behandelt werden:

```
CurCommandSet: TCommandSet = [0..255] = [cmZoom,
cmClose, cmResize, cmNext];
```

Querverweis

cmXXXX-Konstanten, *TView.DisableCommands*, *TView.EnableCommands*, *TView.GetCommands*, *TView.SetCommands*.

TDrawBuffer-Typ

VIEWS

Deklaration

```
TDrawBuffer = array[0..MaxViewWidth-1] of Word;
```

Funktion

Puffer dieses Typs werden von den *Draw*-Methoden vieler *View*-Objekte bei der Bildschirmausgabe benutzt. Die auszugebenden Daten werden zusammen mit dem Bildschirmattribut in den Puffer kopiert und dann auf dem Bildschirm ausgegeben:

```
var
  B: TDrawBuffer;
begin
  MoveChar(B, ' ', GetColor(1), Size.X);
  WriteLine(0, 0, Size.X, Size.Y, B);
end;
```

Querverweis

TView.Draw, *MoveBuf*, *MoveChar*, *MoveCStr*, *MoveStr*

TEvent-Typ

DRIVERS

Deklaration

```
TEvent = record
  What: Word;
  case Word of
    evNothing: ();
    evMouse: (
      Buttons: Byte;
      Double: Boolean;
```

```
      Where: TPoint);
    evKeyDown: (
      case Integer of
        0: (KeyCode: Word);
        1: (CharCode: Char;
            ScanCode: Byte));
    evMessage: (
      Command: Word;
      case Word of
        0: (InfoPtr: Pointer);
        1: (InfoLong: Longint);
        2: (InfoWord: Word);
        3: (InfoInt: Integer);
        4: (InfoByte: Byte);
        5: (InfoChar: Char));
end;
```

Funktion Dieser variante Recordtyp speichert sowohl Ereignisse aus den externen Quellen Maus und Tastatatur, als auch Befehlsereignisse, die von Objekten generiert werden.

Querverweis *evXXXX*-Konstanten, *Txxxx.HandleEvent*, *TView.GetKeyEvent*, *GetMouseEvent*

TItemList-Typ

OBJECTS

Deklaration

```
TItemList = array[0..MaxCollectionSize - 1] of Pointer;
```

Funktion Ein Array generischer Zeiger, das von *TCollection*-Objekten benutzt wird.

Querverweis *TCollection*, *PItemList*

TMenu-Typ

MENUS

Deklaration

```
TMenu = record
  Items: PMenuItem;
  Default: PMenuItem;
end;
```

Funktion Dieser Datentyp repräsentiert eine Ebene innerhalb einer Hierarchie von Menüs und Untermenüs. Das Feld *Items* zeigt auf eine Liste von *TMenuItems* und *Default* auf den gerade eingestellten Eintrag, also jene Option in dieser Liste, auf die bei der Ausgabe der Leuchtbalken zeigt. Ein *TMenuView*-Objekt besitzt das Feld *Menu*, das auf eine Variable des Typs *TMenu* zeigt. Die Routinen *NewMenu* und *DisposeMenu* erledigen die Initialisierung bzw. Freigabe von *TMenu*-Records.

Querverweis *TMenuView*, *TMenuItem*, *NewMenu*, *DisposeMenu*, *TMenuView.Menu*

TMenuItem-Typ

MENUS

Deklaration

```
TMenuItem = record
  Next: PMenuItem;
  Name: PString;
  Command: Word;
  Disabled: Boolean;
  KeyCode: Word;
  HelpCtx: Word;
  case Integer of
    0 : (Param: PString);
    1 : (SubMenu: PMenu);
  end;
end;
```

Funktion Dieser Typ repräsentiert einen Menüeintrag, der entweder eine Menüoption, ein Untermenü oder eine Trennlinie sein kann. *Next* zeigt auf den nächsten *TMenuItem*-Record in einer Liste solcher Menüeinträge oder ist **nil**, wenn es sich um den letzten Eintrag handelt. *Name* zeigt auf den String mit der Bezeichnung der Menüoption oder ist **nil**, wenn dieser Eintrag eine Trennlinie definiert. *Command* speichert den Befehl, der abzusetzen ist, wenn die Menüoption selektiert wird. *Disabled* ist True, wenn die Option verfügbar ist, andernfalls False. *KeyCode* enthält den Tastencode des mit dieser Option verbundenen Tastenkürzels oder 0, wenn kein solches Tastenkürzel definiert ist. Über *HelpCtx* können Sie festlegen, welcher numerische Wert ermittelt wird, wenn der Benutzer bei aktivem Menü Hilfe anfordert. Handelt es sich um einen normalen Eintrag, zeigt *Param* auf einen Parameter-String, der innerhalb des Menüfensters rechts ausgegeben wird, oder ist **nil**, wenn kein solcher String definiert ist. Ist der Eintrag ein Untermenü, zeigt *SubMenu* auf einen Record des Typs *TMenu.* Records des Typs *TMenuItem* werden über die Funktionen *NewItem*, *NewLine* und *NewSubMenu* initialisiert.

Querverweis *TMenu*, *TMenuView*, *NewItem*, *NewLine*, *NewSubMenu*

TMenuStr-Typ

MENUS

Deklaration

```
TMenuStr = string[31];
```

Funktion Dieser Stringtyp wird von *NewItem* und *NewSubMenu* benutzt.

Die Beschriftung von Menüoptionen darf die Länge von 31 Zeichen nicht überschreiten.

Querverweis *NewItem, NewSubMenu*

TPalette-Typ VIEWS

Deklaration
```
TPalette = String;
```

Funktion Datentyp zur Speicherung der Farbpaletten.

Querverweis *Txxxx.GetPalette*

TScrollChars-Typ VIEWS

Deklaration
```
TScrollChars = array[0..4] of Char;
```

Funktion Dieses Array speichert die Zeichen, die zur Darstellung eines Rollbalkens benutzt werden.

Querverweis *TScrollBar*

TSItem-Typ DIALOGS

Deklaration
```
TSItem = record
  Value: PString;
  Next: PSItem;
end;
```

Funktion Dieser Recordtyp erlaubt den Aufbau einer einfach verketteten Liste von Strings. Solche Listen sind dort besonders nützlich, wo die volle Funktionalität einer String-Kollektion nicht gebraucht wird (ein Beispiel zeigt die Methode *TCluster.Init*). Mittels der Funktion *NewSItem* können einer solchen Liste neue Einträge angefügt werden.

TStatusDef-Typ MENUS

Deklaration
```
TStatusDef = record
  Next: PStatusDef;
  Min, Max: Word;
  Items: PStatusItem;
end;
```

Funktion Dieser Typ repräsentiert eine Statuszeile. *Next* zeigt auf den nächsten *TStatusDef*-Record in einer Liste solcher Einträge oder

ist **nil**, wenn es sich um den letzten Eintrag handelt. *Min* und *Max* definieren einen Bereich von Hilfenummern, wodurch die Programmkontexte festgelegt werden, innerhalb derer diese Statuszeile auszugeben ist. *Items* schließlich zeigt auf eine Liste von Optionen, die in der Statuszeile anzuzeigen sind, oder ist **nil**, wenn keine Option definiert ist.
Jedes *TStatusLine*-Objekt (also das Objekt, das sich als Statuszeile darstellt) definiert einen Zeiger auf eine Liste von *TStatusDef*-Records, und wählt die Statuszeile, in deren durch *Min* und *Max* festgelegten Bereich die Hilfenummer des aktuellen Programmkontextes enthalten ist. Ein Turbo-Vision-Programm gibt die Statuszeile entsprechend dem gerade aktiven Programmkontext immer wieder neu aus, indem aus *TProgram.Idle* die Methode *TStatusLine.Update* aufgerufen wird.
Records des Typs *TStatusDef* werden über die Funktion *NewStatusDef* initialisiert.

Querverweis *TStatusLine*, *TProgram.Idle*, *NewStatusDef*

TStatusItem-Typ

MENUS

Deklaration

```
TStatusItem = record
  Next: PStatusItem;
  Text: PString;
  KeyCode: Word;
  Command: Word;
end;
```

Funktion

Dieser Typ repräsentiert eine Option in der Statuszeile, die aber nicht unbedingt dort angezeigt wird. *Next* zeigt auf den nächsten *TStatusItem*-Record in einer Liste solcher Einträge oder ist **nil**, wenn es sich um den letzten Eintrag handelt. *Text* zeigt auf einen String, der diese Option beschreibt, zum Beispiel »**Alt-X** Exit«, oder ist **nil**, wenn die Option nicht angezeigt werden soll. In diesem Fall definiert ein solcher Eintrag einfach ein Tastenkürzel. *KeyCode* enthält den Tastencode des mit dieser Option verbundenen Tastenkürzels oder 0, wenn kein solches Tastenkürzel definiert ist. *Command* speichert den Befehl, der abzusetzen ist, wenn die Option selektiert wird.
Records des Typs *TStatusItem* werden nicht nur dazu benutzt, Optionen in der Statuszeile zu zeigen, sondern dienen auch dazu, Tastenkürzel zu definieren, also Tastencodes mit bestimmten Befehlen zu verbinden. Die Methode *TProgram.GetEvent* ruft für alle Ereignisse des Typs *evKeyDown* *TStatusLine.HandleEvent* auf. Diese Methode durchsucht die mit der aktuellen Statuszeile verbundene Liste von *TStatusItem*Records nach einem Eintrag, der den anliegenden Tastencode enthält.

Wird ein solcher Eintrag gefunden, konvertiert *HandleEvent* das Ereignis des Typs *evKeyDown* in ein Ereignis des Typs *evCommand*, wobei der Wert von *Command* aus dem *TStatusItem*-Record übernommen wird. Records des Typs *TStatusItem* werden über die Funktion *NewStatusKey* initialisiert.

Querverweis *TStatusLine, NewStatusKey, TStatusLine.HandleEvent*

TStreamRec-Typ

OBJECTS

Deklaration

```
PStreamRec = ^TStreamRec;
TStreamRec = record
  ObjType: Word;
  VmtLink: Word;
  Load: Pointer;
  Store: Pointer;
  Next: Word;
end;
```

Funktion

Jeder Objekttyp in Turbo Vision muß registriert werden, bevor dieses Objekt in einem Stream geschrieben oder aus einen Stream gelesen werden kann. Ein *TStreamRec*-Record speichert die dem zu registrierenden Objekt entsprechenden Daten; dieser Record wird dann der Prozedur *RegisterTypes* übergeben.

Die Felder dieses Registrierungsrecords haben folgende Funktion:

Feld	Funktion
ObjType	Ein zur Identifizierung des Objekttyps bestimmter eindeutiger numerischer Wert
VmtLink	Stellt die Verbindung zur VMT dieses Objekttyps her
Load	Ein Zeiger auf den Konstruktor *Load* dieses Objekttyps
Store	Ein Zeiger auf die Methode *Store* dieses Objekttyps
Next	Verweist auf den nächsten *TStreamRec*-Record

Turbo Vision reserviert für den eigenen Gebrauch den Bereich von 0 bis 999 zur Registrierung der Standardobjekte. Der Breich von 1000 bis 65535 steht dagegen zur Verfügung.
Bei der Namensgebung folgt Turbo Vision der Konvention, den *TStreamRec*-Record für das Objekt *Txxxx Rxxxx* zu nennen. Damit würde dem Objekt *TCalculator* ein Record *RCalculator* zugeordnet werden, wie der folgende Programmausschnitt zeigt:

```
type
  TCalculator = object(TDialog)
    constructor Load(var S: TStream);
    procedure Store(var S: TStream);
    ...
  end;

const
  RCalculator: TStreamRec = (
    ObjType: 2099;
    VmtLink: Ofs(TypeOf(TCalculator)^);
    Load: @TCalculator.Load;
    Store: @TCalculator.Store);

begin
  RegisterType(RCalcualtor);
  ...
end;
```

Querverweis *RegisterType*

TStrIndex-Typ

OBJECTS

Deklaration `TStrIndex = array[0..9999] of TStrIndexRec;`

Funktion Wird von *TStringList* und *TStrListMaker* genutzt.

TStrIndexRec-Typ

OBJECTS

Deklaration
```
TStrIndexRec = record
  Key, Count, Offset: Word;

end;
```

Funktion Wird von *TStringList* und *TStrListMaker* genutzt.

TSysErrorFunc-Typ

DRIVERS

Deklaration
```
TSysErrorFunc = function(ErrorCode: Integer;
                         Drive: Byte): Integer;
```

Funktion Der Funktiostyp *TSysErrorFunc* wird von der Abfangroutine für kritische Fehler verwendet, und gibt eine Meldung aus, die über *ErrorCode* aus einer entsprechenden Liste ausgewählt wurde. Der Parameter *Drive* übermittelt das Laufwerk, das möglicherweise für die Fehlersituation ursächlich war. Die

entsprechende Standardroutine in Turbo Vision ist die Funktion *SystemError*.

Querverweis *SysErrorFunc*, *SystemError*, *InitSysError*, *DoneSysError*

TTerminalBuffer-Typ TEXTVIEW

Deklaration
```
TTerminalBuffer: array[0..65534] of Byte;
```

Funktion Ein Puffer dieses Typs wird *TTerminal*-Objekten zugeordnet.

Querverweis *TTerminal*

TTitleStr-Typ VIEWS

Deklaration
```
TTitelStr: string[80];
```

Funktion Dieser Typ wird dazu benutzt, einem Fenster einen Titel zuzuordnen.

TVideoBuf-Typ VIEWS

Deklaration
```
TVideoBuf: array[0..3999] of word;
```

Funktion Dieser Datentyp dient dazu, den Inhalt des Bildschirmspeichers in einer Variablen dieses Typs abzulegen.

Querverweis *InitVideo, TGroup.Buffer*

TWordArray-Typ OBJECTS

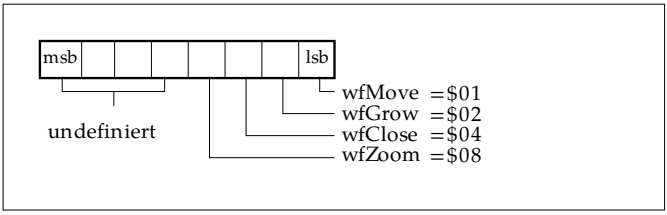
Deklaration
```
TWordArray = array[0..16383] of word;
```

Funktion Dieser Datentyp wird von einigen Objekten dazu benutzt, über *GetMem* und einen Zeiger auf solch ein Array Speicher der jeweils notwendigen Größe zu reservieren.

wfXXXX-Konstanten VIEWS

Funktion Jedes *TWindow*-Objekt definiert das Feld *Flags*, dessen einzelne Bits darüber Auskunft geben, welche Eigenschaften dieses Objekt besitzt.

Werte Die Konstanten, deren Namen mit *wf* beginnen, dienen dazu,



dem Feld *Flags* die gewünschten Werte zuzuweisen, und sind folgendermaßen definiert:

Konstante	Wert	Bedeutung
wfMove	$01	Das Fenster kann einschließlich dessen oberster Zeile über den Bildschirm bewegt werden.
wfGrow	$02	Auf dem Rahmen des Fensters ist ein Icon sichtbar, das mit der Maus angeklickt werden kann, um die Größe der Fensters zu ändern.
wfClose	$04	Der Fensterrahmen besitzt ein Schließfeld, mit dem das Fenster geschlossen werden kann, wenn es mit der Maus angeklickt wird.
wfZoom	$08	Der Rahmen des Fensters ist mit einem Zoomfeld versehen, das mit der Maus angeklickt werden kann, um das Fenster auf Bildschirmgröße zu vergrößern.

Querverweis *TWindow.Flags*

wnNoNumber-Konstante

VIEWS

Deklaration `wnNoNumber = 0;`

Funktion Wenn diese Konstante dem Feld *Number* eines *TWindow*-Objekts zugeordnet wird, bedeutet dies, daß dem Fenster keine Nummer zugewiesen ist, und dieses Fenster daher auch nicht mit der Tastenkombination `[Alt][Nummer]` selektiert werden kann. Dies ist nur dann möglich, wenn das Feld *Number* einen Wert enthält, der zwischen 1 und 9 liegt.

Querverweis *TWindow.Number*

WordRec-Typ

OBJECTS

Deklaration

```
WordRec = record
  Lo, Hi: Byte;
end;
```

Funktion Dieser Recordtyp ermöglicht den Zugriff auf das niederwertige (*Lo*) und das höherwertige (*Hi*) Byte eines Word.

Querverweis *LongRec*

wpXXXX-Konstanten

VIEWS

Funktion Über diese Konstanten kann eine der drei vordefinierten Farbpaletten für ein *TWindow*-Objekt ausgewählt werden. Gemäß der Voreinstellung benutzt ein *TWindow*-Objekt die Palette *wpBlueWindow*, ein *Dialog*-Objekt die Palette *wpGrayWindow*.

Werte

Konstante	Wert	Bedeutung
wpBlueWindow	0	Text in gelb vor blauem Hintergrund
wpCyanWindow	1	Text in blau vor türkisfarbenem Hintergrund
wpGrayWindow	2	Text in schwarz vor grauem Hintergrund

Querverweis *TWindow.Palette, TWindow.GetPalette*

Index

A
Aktionsschalter 63, 97, 142
  aktiver 98
  fokussierter 98
  Funktionsweise 66
Anweisung
  with 11, 20
  with (implizit) 16
Arbeitsfläche 75
Auswahlfeld
  Markierungsfeld 99
  Schaltfeld 99

B
Befehl
  aktivieren 167
  Benennung 76
  deaktivieren 167
  definieren 77, 166
  in Turbo Vision definierte 76
  reservierte Werte 167
  TCommandSet 167
  Werte 167
Beispielprogramme
  FigureDemo 44
  FIGURES.PAS 39
Bereichsfehler
  virtuelle Methoden 34
Bezeichner
  Konstantenpräfixe 228
  Konventionen 228
Bildschirmmüll 86
Bindung
  dynamische 29, 35
  statische 29
Bit
  Additionsoperator 224
  and-Operator 225
  Bitmaske 224
  Flag 223
  höherwertiges 224
  löschen 223, 225
  niederwertiges 224
  not-Operator 225
  Operation 224
  or-Operator 224 f
  setzen 223 f

C
C++ 6
ClearEvent 160, 175
Compileranweisung
  $R (Prüfung virtueller Methoden) 34
Computer-Simulation 21
constructor
  Schlüsselwort 34

D
DataSize 103
Datei
  nicht typisierte 195
Deklaration
  Objektinstanz 10
Desktop 62
Destruktor
  Definition 49
  Deklaration 49
  Freigabe dynamischer Objekte 51
  polymorphe Objekte 50
  statisch oder virtuell 49
  überschreiben 118
Dialogfenster 143
Dienstverhältnis 139
Dispose (Standardprozedur)
  erweiterte Syntax 48
DisposeStr 186
DragMode 145, 149
Draw-Methode 85

über Puffer 87
überschreiben 85, 92
dynamische Bindung 29
mit polymorphen Objekten 35

E
Eingabefeld 101
beschriften 102
Fähigkeiten 102
initialisieren 102
mit Gedächtnis 107
Elemente von Turbo Vision 61
Ereignisse 61
stumme Objekte 62
View-Objekte 61
Ereignis 133
abgearbeitetes 160
Adressaten 162
bearbeites 170
Bearbeitung 167 f
Befehl 161, 166, 172
Beschaffung 171
Botschaft 160, 174
ClearEvent 160, 175
EventError 161, 170
EventMask 163
evXXXX-Konstanten 159, 169
fokussiertes 162, 164 f
geklautes 220
GetEvent 161, 171
Gültigkeitsbereich 145
Idle 172
Kategorien 159
liegengelassenes 170
Maus 159
Message 174
ofPostProcess 164
ofPreProcess 164
Phase 164 f
Positions- 162
Quelle 158
quittieren 174
Rundruf 163
Tastatur 160
Weitergabe 161 ff
Ereignis-Record 133, 158, 168, 174
InfoPtr 160, 170
Ereignisbehandlung 39
Ersatzmethode 23
Erweiterbarkeit 44
EventMask 220
exportierte Objekttypen 18

F
Farbpalette 151
ändern 153 f
Einträge 152
erweitern 155
GetColor 153
Indizierung 152
Feld 17
Gültigkeitsbereich 15
Objekt 11
privates 14 f
Vererbung 11
Zugriff 11, 14, 21
Fenster
Nummern 81
wnNoNumber 81
FigureDemo 44
FIGURES.PAS 39
Fokus 143
Select 98
SelectNext 98
versetzen 98
Fokussierung 142
Funktion
lokale 188

G
GetClipRect 89
GetData 103
GetEvent 161
GetHlpCtx 176
GrowMode 145, 148
Gruppe 81, 134
Ausgabe 137
Daten übergeben 103
Insert 81
Objekte einfügen 135
Z-Ordnung 135
Gruppen 121
Gültigkeitsbereich 17
Felder 15
Methode 15, 17
Objekt 15

H
HandleEvent-Methode 133, 168, 220
überschreiben 118
hcNoContext 78
Hierarchie
Objekt 10
Hierarchie der Objekte 110
Hilfenummer 176

Hilfesystem 175
  hcNoContext 78
  kontextsensitives 78

I
Instanz
  Definition 8
  Objekt- 10

K
Kapselung 5
Kollektion 127, 213
  Anfangsgröße 184
  Daten speichern 185
  dynamische Größe 184
  Einsatz 185
  im Stream speichern 204
  polymorphe 184
  polymorphe Objekte speichern 191
  sortierte 189
  Speicherfreigabe 191
  Speicherung in einer - 186
  Speicherverwaltung 194
  Strings speichern 190
  Typprüfung 184
  Vorteile 183
  Wiederholmethode 187 f, 193
Kompatibilität
  Objekte 29 f
  Zeiger auf Objekte 30
Konstruktor 47
  Definition 34
  und virtuelle Methoden 34
  VMT 34
Konstruktor-Methode
  Felder initilalsieren 116
  überschreiben 116 f
Koordinantensystem 119, 132
  Ursprung 131
Koordinaten
  globale 119
  lokale 119

L
Liste, verkettete 51
LowMemory 178 f

M
Markierungsfeld 99
  beschriften 101
  Initialisierung 99
  Optionen 100
MaxViewWidth 87
Menüleiste 63, 75
  Initialisierung 77
Message 174
Methode
  abstrakte 113
  Assembler 17
  Aufruf 13
  Bezeichner 15
  Definition 12
  Deklaration 13 ff
  ersetzen 23
  extern 17
  Gültigkeitsbereich 15, 17
  lokale Variable 17
  oder Prozedur 38
  Parameter 17
  Parameterbezeichnung 17
  Position in der Hierarchie 38
  private 14 f
  Probleme mit statischen 26
  pseudo-abstrakte 113
  Self-Parameter 16
  statisch 27, 113
  überschreiben 115 ff
  virtuell 28, 71, 113
  vollständiger Bezeichner 20
  Zugriff auf Feld 20
Methodenbezeichner
  vollständiger 13, 15
  Zugriff auf Objektfelder 20
Methodentabelle (VMT) 196
modaler Dialog 96
Modalität 96, 145
  abgeben 97
Müll
  auf dem Bildschirm 86

N
Nachkomme 9
  direkter 10
New (Standardprozedur) 47
  als Funktion 48
  erweiterte Syntax 47
NewItem 78
NewStatusDef 76
NewStatusKey 76
NewStr 186
NewSubMenu 78

O
object

Schlüsselwort 10
Objekt
ableiten 112, 115
abstraktes 110 f
Basisobjekt 110
Definition 6
Deklaration von Destruktoren 49
Destruktoren und polymorphe 50
Dienstverhältnis 139
dynamische Freigabe mit
Destrutkoren 51
dynamische Instanz 47, 51
dynamisches 114
Erweiterbarkeit 44
Erweiterung 23
Felder 11
fokussiertes 142, 144
Gruppe 81, 94, 121
Gültigkeitsbereich 15
Hierarchie 10, 110, 138
Identifikationsnummer 203 f, 206
Instanz von 111
Kompatibilität 29
Kompatibilität von Parametern 31
Konstruktor 34
Load-Konstruktor 199
modales 96
Nachkomme 9
polymorph 31
registrieren 196, 200, 202
selektiertes 142, 144
statische oder virtuelle Destruktoren 49
Store-Methode 199
Stream definieren 211
stummes Objekt 62, 110, 125
Vererbung 11
View-Objekt 79, 110, 120
virtuelle Methodentabelle 34
VMT und Konstruktoren 34, 47
Vorfahr 9
Zugriff 11, 14, 21
Objektfelder
Zugriff 21
Objekthierarchie 10, 110
gemeinsame Eigenschaften 38
Objektinstanz 111
Deklaration 10
dynamische 51
Objekttyp
exportiert 18
Kompatibilität 29
Objektzeiger
Kompatibilität 30
ofPostProcess 105, 164 f
ofPreProcess 164 f
Operation
atomare 177, 179
komplexe 177
Options 145

P
Parameter
Bezeichnung in Methoden 17
Self 16
Phase
phFocused 165
phPostProcess 165
phPreProcess 165
polymorphe Objekte 31
dynamische Bindung und 35
virtuelle Methoden 33
Polymorphie 5, 28 ff
private-Deklaration 15
Prozedur
lokale 187
oder Methode? 38
Puffer-Routinen 88
Punktmethode 11, 20

R
RegisterApp 202
RegisterColor 202
RegisterDialogs 202
RegisterMenus 202
RegisterObjects 202
RegisterType 200, 217
RegisterViews 203
Registrierungsnummer 203 f, 217
Registrierungsprozeduren 203
Registrierungsrecord 204, 206, 217
für Standardobjekte 202
Rekord
Typ 8
Ressourcen-Datei 213 f
anlegen 214
Einträge lesen 216
Schlüssel 213, 216 f
Strings 217
Rollbalken 91, 93

S
Schaltfeld 99
beschriften 101
Initialisierung 100

Optionen 100
Schlüsselworte
constructor 34
Object 10
virtual 33
Selektierungspfad 143
Selektion 142
Self, Parameter 16
SetData 103
SetState 150
Sicherheitszone 178 ff
Simula-67 21
Simulation 21
SizeOf 206
Smalltalk 6
Speicherfresser 181
Speicherplatzreservierung 47
Freigabe mit Destruktoren 51
Speicherverwaltung
LowMemory 178
Sicherheitszone 178 ff
Speicherfresser 181
ValidView 179
Standardbefehle 76
Standardobjekt
registrieren 202
Standardprozeduren
Dispose (erweiterte Syntax) 48
New (erweiterte Syntax) 47
New als Funktion 48
State 150
statische Bindung 29
statische Methoden 27
Probleme mit dem Gültigkeitsbereich 26
Statuszeile 63, 75
Initialisierung 77
Stream 195, 213
Arbeitsfläche speichern 209
Daten schreiben 205
Daten speichern 211
definieren 211
Fehler 199, 212
GetPeerViewPtr 209
GetSubViewPtr 208 f
Gruppe speichern 198
initialisieren 197
Kollektion speichern 204
kopieren 210
Objekt laden 198, 203
Objekt registrieren 199
Objekt speichern 198, 203
Objekte 125
PutPeerViewPtr 209
PutSubViewPtr 208 f
RegisterType 200
Registrierung der Standardobjekte 202
Registrierungsnummer 201
Registrierungsrecord 200
sequentieller Zugriff 197
Standardobjekte registrieren 207
Store-Methode 203 ff
TBufStream 126
TDosStream 126
TEmsStream 126
TStream 125, 197
wahlfreier Zugriff 197, 210 f
Zeiger speichern 207
Subview 134

T

TApplication 68, 73, 121, 134
HandleEvent 69
Init 68, 75
InitMenuBar 69
Tastenkürzel 165
globale 105
lokale 105
mittels Tilde-Zeichen definieren 76
Tücken 105
TBufStream 126, 197
TButton 97, 122
bfDefault 98
bfNormal 98
initilisieren 98
TCheckBoxes 111, 123
TCluster 111, 123
TCollection 116, 124, 207
FirstThat 188
ForEach 187
LastThat 188
Vorteile 183
TDesktop 121
TDialog 122
TDosStream 126, 197
TDrawBuffer 87 f
TEmsStream 126
TEvent 158, 166, 168 f
InfoPtr 160, 170
What 159
TFrame 122
TGroup 121, 134, 137
TGroup.Phase 165
THistory 107, 123

THistoryViewer 123
THistoryWindow 123
Tilde-Zeichen 76, 101
TInputLine 101, 123
TLabel 101, 125
TListBox 107, 124
TListViewer 107, 123
TMenuBar 123
TMenuBox 123
TMenuView 123
TObject 116, 120, 185
TPoint 119, 131
TProgram 121
TRadioButtons 111, 123
TRect 120, 131
TResourceCollection 127
TResourceFile 126, 214
TScrollBar 112, 124
TScroller 90, 124
  Rollbalken 91
TSortedCollection 127, 189, 190
  Compare 189 f
  KeyOf 189 f
TStaticText 106, 125
TStatusLine 125
TStream 116, 125, 197 f, 211
  Write 205
TStringCollection 127, 185
TStringList 128, 217
TStrListMaker 128, 217
TTerminal 124
TTextDevice 124
TView 83, 130
  Darstellung 85
  Draw 85
  DrawView 85
  GetClipRect 89
  GetData 103
  Modalität 96
  Rahmen 83
  SetData 103
  WriteBuf 88 f
  WriteChar 88
  WriteLine 88 f
  WriteStr 88
TWindow 122
  Fensternummern 81
  initialisieren 81
  wnNoNumber 81
Typ
  Objekt 18
  Rekord 8

U
Unit
  Objekte exportieren 18

V
ValidView 179 f
Variable
  lokale 17
Vererbung 5
verkettete Liste 51
View-Baum 139
View-Objekt 79, 130, 158
  Anforderungen 130
  Bereich 133
  Draw-Methode 132 f
  EventMask 163
  Farbpalette 154
  fokussiertes 165
  Fokussierung 142
  modales 145
  Option 164
  Selektion 142
  Status 150
  terminales 122
  Ursprung 131
  View-Baum 139
virtual
  Schlüsselwort 33
virtuelle Methoden 28
  Ereignisbehandlung 39
  Konstruktoren und 34
  Prüfung 34
  und polymorphe Objekte 33
  virtuelle Methodentabelle (VMT) 34, 47
  Zeigerinitialisierung 34
  VMT (virtuelle Methodentabelle) 34
Vorfah r 7, 9
  direkter 10

W
with
  Anweisung 11, 16, 20
wnNoNumber 81
WriteBuf 88 f
WriteChar 88
WriteLine 88 f
WriteStr 85, 87 f

Z
Z-Ordnung 135f, 162, 164

Gestärkt durch die Leistungsfähigkeit und Effektivität der objektorientierten Programmierung gleibt die Stellung von Turbo Pascal als Marktführer auch in Zukunft gesichert... Sicher ist nur eins, OOP wird neue Bereiche eröffnen, sei es in Datenbank-Management-Systemen oder auch in Programmiersprachen der Zukunft.

Jürgen Mertes/Klaus Schäfer
PC Magazin, 18. Oktober 1989

Das objektorientierte Modell hat das Zeug dazu, nach der Einführung der strukturierten Programmierung vor gut zwei Jahrzehnten die zweite »sanfte« Wende auf dem Gebiet der Software-Entwicklung zu sein.

Ronald Fischer
Toolbox, Oktober 1989

B O R L A N D